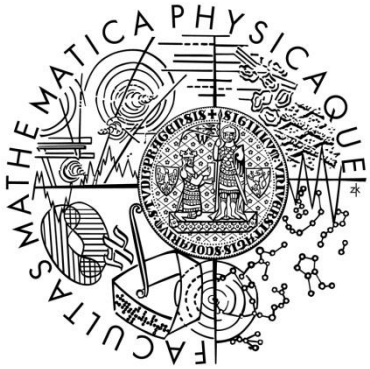
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Jakub Šmíd

Agent optimization by means of genetic programming

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Roman Neruda, CSc.

Study programme: Computer Science

Specialization: Theoretical Computer Science

Prague 2012

Acknowledgment:

I deeply thank my advisor, Roman Neruda, whose help, advice and supervision was invaluable. I also thank Jaroslav Švelch, Dan Kobr and Milan Plachý. Last but not least, I want to thank my parents and all the people who keep supporting me.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

Název práce: Optimalizace agentů prostřednictvím genetického programování

Autor: Bc. Jakub Šmíd

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Roman Neruda, CSc.

E-mail vedoucího: roman@cs.cas.cz

Abstrakt: Tato práce se zabývá problémem výběru nejvhodnějšího agenta pro novou, agenty doposud neviděnou úlohu z oblasti dobývání znalostí. Je navržena metrika na prostoru úloh z oblasti dobývání znalostí a na základě této metriky je vybráno několik nejbližších úloh. Tento výběr je předložen jako vstup programu, který byl vyvinut pomocí genetického programování a který odhaduje výsledky agentů na nové úloze jak z pohledu chybovosti, tak z pohledu časové náročnosti. Je implementován JADE agent poskytující rozhraní umožňující získat výsledky odhadu ostatním agentům v reálném čase.

Klíčová slova: metaučení, genetické programování, multi-agentní systémy

Title: Agent optimization by means of genetic programming

Author: Bc. Jakub Šmíd

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc.

Supervisor's e-mail address: roman@cs.cas.cz

Abstract: This thesis deals with a problem of choosing the most suitable agent for a new data mining task not yet seen by the agents. The metric is proposed on the data mining tasks space, and based on this metric similar tasks are identified. This set is advanced as an input to a program evolved by means of genetic programming. The program estimates agents performance on the new task from both the time and error point of view. A JADE agent is implemented which provides an interface allowing other agents to obtain estimation results in real time.

Keywords: metalearning, genetic programming, multi-agent systems

1 Introduction

Data mining - the process of finding new patterns in data sets - is now widely used in medicine, economics, bioinformatics and other important areas of human interest. Many different algorithms exist and are used for this task of pattern extraction.

According to the *no free lunch theorem* stated in [1], the average performances of data mining algorithms on all data mining problems are equal. That means that elevated performance of any algorithm over one class of problems is paid for in performance over another class. Therefore, the key to success when dealing with a data mining problem is in binding the problem with an algorithm having elevated performance on the class of the problem. Because many fields depend on data mining techniques it is crucial to propose and improve such bindings.

Data mining problems have many parameters (problem type, number of instances, number of attributes, types of attributes, etc.), and they may be compared based on these parameters. One can assume that similar problems belong to the same class and that the performance of any algorithm will be similar on the problems of that class (that means problems with similar parameters). Having a new data mining problem, estimated performance of all algorithms in question may be calculated by utilizing performance of those algorithms on similar problems with the one at hand. The algorithms with the best estimation would be suggested candidates for solving the problem at hand.

This thesis proposes a method that estimates performance of algorithms on a given data mining problem. The method is implemented in the form of a software agent exposing an interface which returns evaluation results in real time. With more performance results available, the estimation accuracy is expected to rise.

The method core is based on metalearning. Metalearning has the ability to improve performance over time, so it is the ideal candidate for our needs. The metalearning approach used utilizes an estimation function that is evolved by a machine learning method called genetic programming.

The structure of this thesis is the following. Chapter 2 defines the terms necessary to formulate the goals of the thesis more precisely. Related work that has dealt with similar problem/s in the past is also discussed. Chapter 3 provides overview of computational intelligence methods that will be used further in the thesis. Chapter 4 lists data available from machine learning experiments, and proposes a method that can utilize this data to estimate other algorithms performance on the given task. Architecture of an agent encapsulating the method is proposed. Chapter 5 describes technologies used to implement the metalearning method and the agent proposed in chapter 4. Chapter 6 describes experiments performed with the method proposed in 4. Chapter 7 discusses the results achieved in this work and concludes the thesis. Chapter 8 suggests future improvements of the methods and experiments proposed in this thesis.

2 Problem analysis

In this chapter we will first state definitions crucial to our thesis – definition of the agent, the intelligent agent, and metalearning. The goal of this thesis is specified using these terms. Related work is discussed afterwards.

2.1 Agent

There is no common agreement on what an *agent* is because this term is used in many fields of computer science. Every field has different objectives for agents, so the definitions are tailored to fulfill those purposes. In this thesis we will use definition stated in [2] that is suitable for our needs: An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.

In this definition, the agent is part of an environment – it receives sensory input from the environment, and alters it by its own actions. There are many different properties of an environment. For our purposes distinguishing single and multi-agent (environment is populated by more than one agent changing the environment) environment is sufficient, for the list of other properties, see [3]. Autonomy in this thesis means that agents are able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behavior.

2.2 Intelligent agent

In this thesis we will also use the definition of *intelligent agent* stated in [2], which is suitable for our needs: an intelligent agent is one that is capable of flexible autonomous action in order to meet its design objectives, where flexibility means three things:

- reactivity: intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives.
- pro-activeness: intelligent agents are able to exhibit goal-directed behavior by taking the initiative in order to satisfy their design objectives.

- social ability: intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

Speech act theory [4] is often used as the model for communication among computational agents. This theory views human natural language as actions that results in changes in the internal state of the recipient. Verbal actions of this kind are called speech acts. A speech act has three aspects:

- Locution: the physical utterance by the speaker.
- Illocution: the meaning of the utterance intended by the speaker.
- Perlocution: the action that results from the locution.

In communication among humans, the intent of the message is not always easily identified. Howewer, for communication among agents, we want to ensure that there is no doubt about the type of message. Speech act theory uses the term *performative* to identify the illocutionary force of utterance. Illocutionary force can be broadly classified as assertives (statements of fact), directives (commands), commissives (commitments), declaratives (statements of fact), and expressives (expressions of emotion).

2.3 Meta learning

In the context of machine learning, *metalearning* is the process of learning to learn [5]. Metalearning differs from base-learning in the scope of the level of adaptation; whereas learning at the base-level is focused on accumulating experience on a specific learning task (e.g., credit rating, medical diagnosis, mine-rock discrimination, fraud detection, etc.), learning at the meta-level is concerned with accumulating experience on the performance of multiple applications of a learning system. If a base-learner fails to perform efficiently, one would expect the learning mechanism itself to adapt in case the same task is presented again. Briefly stated, the field of meta-learning is focused on the relation between tasks or domains and learning strategies. In that sense, by learning or explaining what causes a learning system to be successful or not on a particular task or domain, we go beyond the goal of producing more accurate learners to the additional goal of

understanding the conditions (e.g., types of example distributions) under which a learning strategy is most appropriate [6].

One common approach to metalearning is based on the idea that base-level learning algorithm will perform similarly on problems from the same class. Meta learning based on this approach divides the computation into two steps. The first one (often called *zooming*) identifies the set of similar already computed tasks to the task at hand. The second part utilizes the information of data mining algorithms' performance on those sets. This approach will be used in our work. Every data mining algorithm will correspond to some agent (encapsulating the algorithm).

2.4 Goals statement

By defining all the necessary terms, the concrete goals of this thesis can be now stated:

- Propose a metric comparing data mining problems. Similar set of tasks can be identified by the metric proposed.
- Identify a search space for estimation of agent performance on the given task. This search space will include variables from similar tasks set.
- Design an optimization method for searching the space. The goal of this optimization method is to yield a method capable of estimating the agent's performance.
- Implement this optimization method and run experiments to tune the optimization method and to obtain the estimation method.
- Propose an architecture of an intelligent agent encapsulating similar set identification (*zooming*) and estimation method. This agent will be called *Estimation agent.* The goal of the architecture is to expose an interface allowing other agents in the multi-agent environment to obtain estimation results.
- Implement the proposed Estimation agent architecture in an multi-agent environment.

2.5 Related work

2.5.1 Zoomed Ranking: Selection of Classification Algorithms Based on Relevant Performance Information

Authors of [7] employ the k-nearest neighbor algorithm with a distance function based on a set of statistical, information theoretic and other dataset characterization measures to implement *zooming*. In the second phase, a ranking on the basis of the performance information of the candidate algorithms on the selected datasets is constructed. The *adjusted ratio of ratios* ranking method is presented which processes performance information based on accuracy and time.

Let us first describe the zooming phase. The relevance of a processed dataset to the one at hand is defined in terms of similarity between them, according to meta-attributes. It is given by function:

$$dist(d_i, d_j) = \sum_x \delta(v_{x,d_i}, v_{x,d_j}),$$

where d_i and d_j are datasets, v_{x,d_i} is the value of meta-attribute x for dataset d_i, and $\delta(v_{,d_i}, v_{x,d_j})$ is the distance between the values of meta-attribute x for datasets d_i and d_j. In order to give all meta-attributes the same weight they were normalized in the following way:

$$\delta\left(v_{x,d_i}, v_{x,d_j}\right) = \frac{\left|v_{x,d_i} - v_{x,d_j}\right|}{\max_{k \neq i}(v_{x,d_k}) - \min_{k \neq i}(v_{x,d_k})}.$$

If both datasets do not have certain attribute, distance is 0 for that attribute. If only one of the datasets is missing the attribute distance is then maximal value of 1. All attributes were obtained by Data Characterization Tool. K-nearest neighbor algorithm is then used to identify k cases nearest to the dataset in hand.

Let us focus on the ranking phase now. The adjusted ratio of ratios uses information about accuracy and given execution time to rank the given classification algorithms.

$ARR_{a_p,a_q}^{d_i}$ is defined as:

$$ARR_{a_p,a_q}^{d_i} = \frac{\frac{SR_{a_p}^{d_i}}{SR_{a_q}^{d_i}}}{1 + \frac{\log\left(\frac{T_{a_p}^{d_i}}{T_{a_q}^{d_i}}\right)}{K_T}},$$

where $SR_{a_p}^{d_i}$ and $T_{a_p}^{d_i}$ are the success rate and duration of algorithm a_p on dataset d_i, and K_T is a user-defined value that determines the relative importance of time. The method aggregates the given performance information as follows: first adjusted ratio of ratios table is created for each dataset. The table for dataset d_i is filled with corresponding $ARR_{a_p,a_q}^{d_i}$. Next, a pairwise mean adjusted ratio of ratios is calculated for each pair of algorithms:

$$ARR_{a_p,a_q} = \left(\sum_{d_i} ARR_{a_p,a_q}^{d_i}\right)/n,$$

where n is the number of datasets. This represents an estimate of the general advantage/disadvantage of algorithm a_p over algorithm a_q. Finally, we derive the overall mean adjusted ratio of ratios for each algorithm:

$$ARR_{a_p} = \left(\sum_{a_q} ARR_{a_p,a_q}\right)/(m\ -1),$$

where m is the number of algorithms. The ranking is derived directly from this measure. The higher the value an algorithm obtains, the higher the corresponding rank.

2.5.2 Metalearning in computational MAS

Authors of [8] take similar approach. To implement *zooming*, metadata containing the following information about the task were utilized:

- Number of attributes in data
- Number of instances
- Data type (one of the following – categorical, integer, multivariate)
- Default task type (set by the user, the most common types are classification and regression)

- Missing values (flag whether data contains unknown or unspecified values)

The following metric is defined between two tasks based on their metadata:

$$d(m_1, m_2) = \sum_{i=1}^{n} w_i \times d_i(m_1[i], m_2[i]),$$

where m_1, m_2 are metadata of compared tasks, w_i is a weight of each attribute and d_i is distance between metadata attributes i which depends on the type of the attribute. For computing distance of Boolean and categorical attributes the following formula was used:

$$d_i(v_1, v_2) = \begin{cases} 0; & if\ v_1 = v_2 \\ 1; & otherwise \end{cases}.$$

Missing values are handled as an extra Boolean attribute, v1 and v2 are actual values of the attribute i.

To compute the distance of numerical attributes, the following formula that maps two values $v_1$and $v_2$on the interval ‹0,1›, was used:

$$d_i(v_1, v_2) = \frac{|v_1 - v_2|}{\max\limits_{v \in i} v - \min\limits_{v \in i} v}.$$

In the second part, the suggested agent is chosen by an algorithm shown in Figure 2.1 (written in pseudocode).

Find Best Agent (*newMetadata*)

```
//step 1: chose the nearest file
metadata ← SELECT * FROM tables. metadata
bestDistance ← int. MaxInt
foreach (m in metadata)do:
        if distance(newMetadata, m) < bestDistance do:
                bestDistance ← distance(newMetadata, m)
                nearestMetadata ← m
//step 2: choose the best agent
nearestFileName ← m. FileName
agent ← SELECT * FROM tables. results WHERE
            dataFile = nearestFileName AND errorrate is MINIMAL
return agent
```

Figure 2.1 - Find best agent

3 Methods used

This chapter describes methods and algorithms we used in our work.

3.1 Genetic algorithms

Genetic algorithm is the meta search heuristic based on Charles Darwin's evolution theory [9] and the laws of inheritance inferred from Gregor Mendel's inheritance theory [10]. According to the evolution theory, the following facts hold:

- Every species is fertile enough that if all offspring survived to reproduce the population would grow.
- Despite periodic fluctuations, populations remain roughly the same size.
- Resources such as food are limited and are relatively stable over time.
- Individuals in a population vary significantly from one another, and much of this variation is inheritable.

From these facts the theory infers the following:

- A struggle for survival ensues.
- Individuals less suited to the environment are less likely to survive and less likely to reproduce; individuals more suited to the environment are more likely to survive and more likely to reproduce and leave their inheritable traits to future generations, which produces the process of natural selection.
- This slowly effected process results in populations changing to adapt to their environments, and ultimately, these variations accumulate over time to form new species.

According to the inheritance theory, inherent properties of each organism are encoded in a structure called genotype. Genotype consists of genes. Each gene corresponds to some trait in organism (e.g. color of eyes). Genotype is inferred from parents' genotype by crossing over their genetic material. Genotype may be also altered during organism lifetime by mutation.

Genetic algorithms (GA) were described by John Holland [11], who utilized principles of the evolution and inheritance theory. Given an optimization problem,

GA bias the solution to the problem as an individual. In the original John Holland's work, the individual was binary encoded, but other encodings are suitable as well. A number of individuals form a population. At first, a population of individuals is created (either randomly or by using some known sub-optimal solutions). Individuals are then evaluated based on their ability to solve the problem by the number known as *fitness*. Individuals proceed to next generation with probability proportional to their fitness (this step is known as *selection*). Each generation new individuals are created from random parents in the current population (this step is known as *crossover*) and some individuals in the current population are altered (this step is known as *mutation*). New generations continue to be created until a termination criterion is satisfied (usually conditions on fitness of some individual, average fitness in population, number of generations or time elapsed since the start of the algorithm). A pseudocode of genetic algorithm is shown in Figure 3.1.

Genetic algorithm

$population \leftarrow$ initializePopulation ()

while terminationCriterion NOT satisfied:

    $population \leftarrow$ selection ($population$)

    $population \leftarrow$ crossover ($population$)

    $population \leftarrow$ mutate ($population$)

return bestIndividual(population)

Figure 3.1 - Genetic algorithm

Since genetic algorithms are stochastic and do not guarantee finding an optimal solution, it could be sometimes beneficial to repeat the whole process and take the best individual from all runs.

Mutation, crossover and other steps altering the population are often generalized as *genetic operators*. Genetic operators will be described when applied to one or several individuals; expansion of the operators on the whole population is typically done by applying the operator on each individual or on a sample of individuals from the population.

3.1.1 Selection

Selection determines how many offspring the individuals will have in the next generation. This should be based on fitness - in general, fitter individuals should have more offspring than those less fit. Common approaches to the selection are:

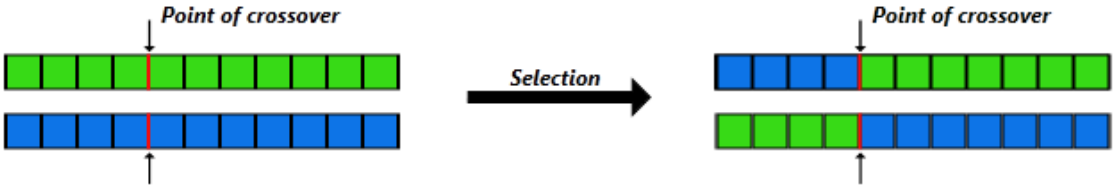
- Roulette selection – let f_k be a fitness of an individual k. Let f_s be the sum of the fitness of all individuals ($f_s = \sum_{i=1}^{n} f_i$), where n is the population size. For each position in the next generation, the roulette is spinnned. In each spin an individual k is selected with probability $\frac{f\mathrm{k}}{fs}$.
- Scaling – same as the roulette selection except that fitness is scaled at the beginning. The most common scaling function is linear function. This can solve some problems in case all individuals have similar fitness (more like random walk) or when there are very large fitness gaps between individuals (high pressure on selecting best individuals).
- Rank based – individuals are sorted by fitness in ascending order. Probability of selection is higher with higher index in the sorted set of individuals.
- Tournament selection – for each position in next generation, a tournament of n-tuple is held. The best individual is selected by the tournament with some fixed probability *p.* If the best individual is not selected, the second best individual is selected with probability *p* and so on. If all previous individuals are not selected, select the worst individual from the tournament.

Selection is often used together with elitism. That means that a few best individuals are automatically selected to the next population (thus we never lose the best solution found so far).

3.1.2 Crossover

Crossover is analogous to reproduction and biological crossover. Two or more individuals (parents) are taken from a population and an offspring is created by combining their features.

In binary coding, crossover is typically implemented as a one point crossover – two parents are selected, then one point in both parents is chosen randomly, and the parts induced by the point chosen are swapped. This creates two offspring



individuals. A more general variant is the n-point crossover, where more points are chosen when creating the offspring. One point crossover is illustrated in Figure 3.2.

Figure 3.2 - Crossover operator

3.1.3 Mutation

Mutation operator alters a part of an individual, thus introducing new features into population. Mutation helps explore those parts of the search space that would be otherwise hard to reach with selection and crossover only. In particular, mutation helps the genetic algorithm get out of the local optima.

In binary coding, mutation is typically implemented in a way that each bit has some small probability $p_{mutation}$ of being flipped. This is ilustrated in Figure 3.3.

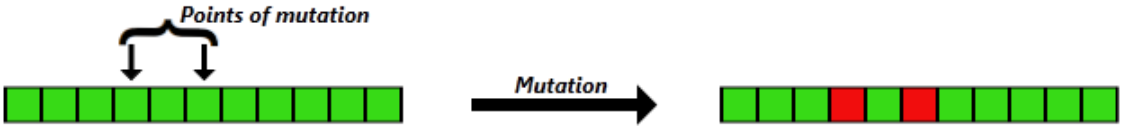


Figure 3.3 - Mutation operator

3.1.4 Other operators

The previous operators are only a small fragment of possible genetic operators. If we have some knowledge about the problem we are trying to solve, we may introduce operators tailored to the problem. Such operators often achieve better results than classic non-specialized operators.

3.2 Genetic programming

Genetic programming (GP) is based on the same idea as genetic algorithms, but the search space is different. Genetic algorithms search the space of possible solutions to the problem, while genetic programming algorithms search the space of programs (hopefully able to solve the problem) instead.

There are many ways of representing a program. Two most common ways of represantation used in genetic programming are [12]:

- Tree representation: tree representation is traditional in genetic programming, and we will use only this representation in this thesis. Inner nodes of the tree represent operators (number of successors of the node equals arity of the operator) and leaves represent operands. Tree structures are easily evaluated and easy to interpret. Genetic operators are also easy to implement, as we will see later in this chapter.
- Linear representation: programs are represented as a sequence of some programming language. Linear representation will not be considered further in this thesis.

The search space is determined by a language *L* consisting of two sets – the function set (having arity greater than zero) and the terminal set (having zero arity). Terminals are either constants or input variables, and they occur only in the leaves of the tree representing the program. Functions are aggregating other functions and terminals, and they occur only in the inner nodes of the tree. The choice of *L* is very important. Program solving the problem have to be encodable in this language - on the other hand, a too complex language will increase the search space exponentially, thus making finding a sufficiently good program nearly impossible. Genetic programming was used successfully in many domains. For some example of domains where GP was used see [12, pp. 111-130].

3.2.1 Initialization

At the beginning of the GP run, each individual in the initial population has to be randomly initialized. This can be achieved using following methods:

- Full method: This method receives an integer specifying the depth of a new individual as an input. This method creates layers sequentially. If the depth of the layer is lower than the target depth, new nodes are created by using functions, otherwise only terminals are used. This method creates a full tree of the target depth.

- Grow method: This method takes an integer specifying maximum depth as an input. New layers are created by using both functions and terminals at random. If maximum depth were to be violated, only terminals are used. By allowing terminals in the inner nodes, the distance between the root node and lists may be less than the specified maximum depth.
- Ramped half-and-half: This method creates half of the new individuals by using the full method and the other half by using the grow method.

3.2.2 Crossover

The principle of the crossover operator is the same as in the original genetic algorithms. Two parents are given and one node from each parent is randomly selected. Subtrees corresponding to these nodes are swapped afterwards. The whole process is illustrated by Figure 3.4. Crossover points are often not selected with uniform probability. Typical GP primitive sets lead to trees with an average branching factor (the number of children of each node) of at least two, therefore the majority of the nodes will be leaves. Consequently, the uniform selection of crossover points leads to crossover operations frequently exchanging only very small amounts of genetic material (i.e., small subtrees); many crossovers may in fact reduce to simply swapping two leaves. To counter this, authors in [13] suggested the widely used approach of choosing functions 90% of the time and leaves 10% of the time.

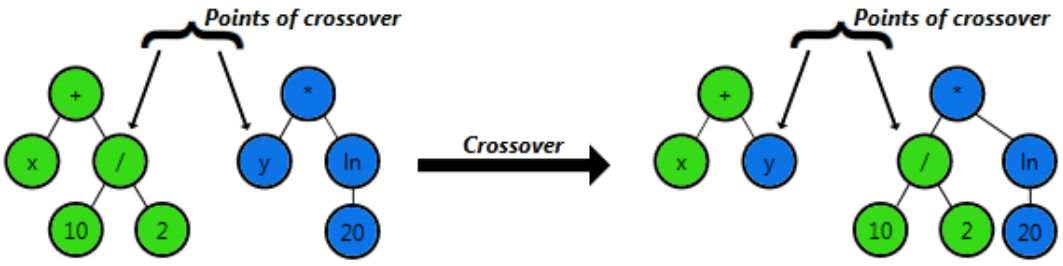


Figure 3.4 - Genetic programming crossover operator

3.2.3 Mutation

Mutation alters part of the tree. A node in the parent is randomly selected. A random tree is initialized by one of the initialization method (see 3.2.1 for the

overview of initialization methods in GP) and the selected node is replaced by the new tree. The whole process is illustrated in Figure 3.5.

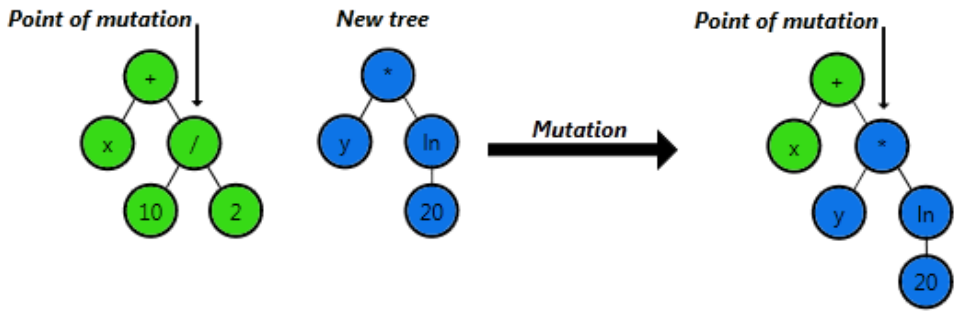


Figure 3.5 - Genetic programming mutation operator

3.2.4 The bootstrap problem

The bootstrap problem may occur in complex domains. When the population is initialized, all individuals often have very low fitness. This happens especially when the ratio of good to bad solutions is very small. It is then hard for the genetic programming algorithm to estimate good places to explore/exploit and the run of the algorithm is similar to the random walk algorithm.

One approach to deal with the bootstrap problem is proposed in [14]. The problem and/or domain is simplified, so there is a better chance that some good individuals are generated during initialization. After good solutions are found for simplified problem/domain we increase the difficulty of the problem, but let the population as it is. We expect that the solutions for simplified problems will not have very low fitness for the more difficult problem (as would probably happen with random initialization). The whole process is repeated until the more difficult problem is equal to our original problem. This approach is called the *incremental evolution*. The incremental evolution was not used in the final proposal of algorithms used in the thesis, although it was considered and may be introduced in the future.

3.2.5 The bloat problem

Bloat refers to a rapid growth of individual sizes without corresponding significant increase of fitness in later generations. In general, software bloat means that a computer program contains features that are never or rarely used. Growth alone could be beneficial, after all we are often searching complex program spaces, but

without the fitness improvement it nearly always is bad. Larger individuals take more time to evaluate, take more space to store, are harder to interpret, and their ability to generalize is greatly reduced.

There are three main theories explaining bloat [12]:

1. Replication accuracy theory states that the success of a GP individual depends on its ability to have offspring that are functionally similar to the parent. As a consequence, GP evolves towards (bloated) representations that increase replication accuracy.
2. Removal bias theory divides nodes in a GP tree into two categories – active code and inactive code. Inactive code is either not executed, or it is executed and its output is then discarded (for example, an inactive code would be a subtree consisting of +(0+0+0+0)). All remaining code is considered active. The theory observes that inactive code in a GP tree tends to be low in the tree, residing, therefore, in smaller-than-average-size subtrees. Crossover events excising inactive subtrees produce offspring with the same fitness as their parents. On average, the inserted subtree is bigger than the excised one, thus such offspring are bigger than average while retaining the fitness of their parent leading ultimately to growth in the average program size.
3. The nature of the program search spaces theory predicts that above a certain size, the distribution of fitness does not vary with size. Since there are more long programs, the number of long programs of a given fitness is greater than the number of short programs of the same fitness. Over a time GP samples longer and longer programs simply because there are more of them.

Techniques were designed to prevent or decrease bloat; few examples of such techniques follow [12]:

1. Size and depth limits. This approach checks after applying genetic operator whether the offspring is beyond the size or depth limit. If it is not, the offspring enters the population. If, instead, the offspring exceeds the limit, one of the parents is returned. Obviously, this implementation does not

allow programs to grow too large. However, there is a serious problem with this way of applying size limits, or more generally, constraints to programs: parent programs that are more likely to violate a constraint will tend to be copied (unaltered) more often than programs that do not. That is, the population will tend to be filled up with programs that nearly infringe the constraint, which is typically not what is desired. The problem can be fixed by not returning parents if the offspring violates a constraint. This can be realized using two different strategies. Firstly, one can just return the oversized offspring, but give it a fitness of 0, so that the selection will get rid of it at the next generation. Secondly, one can simply declare the genetic operation failed, and try again. This can be done in two alternative ways: a) the same parent or parents are used again, but new mutation or crossover points are randomly chosen (which can be done up to a certain number of times before giving up on those parents), or b) new parents are selected and the genetic operation is attempted again.

2. Anti-Bloat genetic operators. This approach modifies genetic operators to reduce the bloat. Among the most recent bloat-control methods are size fair crossover and size fair mutation [15]. These work by constraining the choices made during the execution of a genetic operation so as to actively prevent growth. In size-fair crossover, for example, the crossover point in the first parent is selected randomly, as in standard crossover. Then the size of the subtree to be excised is calculated. This is used to constrain the choice of the second crossover point so as to guarantee that the subtree chosen from the second parent will not be “unfairly” big.
3. Anti-Bloat selection modifies the selection so that bloated individuals have lower probability to be selected into next generation. Tarpeian method [16] controls bloat by acting directly on the selection probabilities in the following equation:

$$E[\mu(t+1) - \mu(t)] = \sum_{l} l \times \big(p(l,t) - \Phi(l,t)\big),$$

where E is the expectation operator, $\mu(t+1)$ is the mean size of the programs in the population at generation $t+1$, l is the program size, $p(l,t)$

is the probability of selecting programs of size l from the population in generation t and $\Phi(l,t)$ is the proportion of programs of size l in generation t. This is done by setting the fitness of randomly chosen longer-than-average programs to 0. This prevents them from being parents. By changing how frequently this is done, the anti-bloat intensity of Tarpeian control can be modulated. An advantage of the method is that the programs whose fitness is zeroed are never executed, thereby speeding up runs. Parsimony pressure method [13] changes the selection probabilities by subtracting a value based on the size of each program from its fitness. Clearly, bigger programs have lower fitness and potentially less offspring under this approach. That is, the new fitness function is:

$$f_{new}(x) = f(x) - c \times l(x),$$

where $l(x)$ is the size of program x, $f(x)$ is its original fitness and c is a constant known as the parsimony coefficient.

4 Algorithm design

This chapter describes the design of all key algorithms proposed to solve the goal of this thesis. We also derive the time complexity of some of these algorithms. First, all available metadata is listed in this chapter. A metric comparing data mining problems utilizing available metadata is proposed. The metric does not depend on the order of attributes like some other metrics used in metalearning. Available performance results of computational experiments that serve as the training data for our approach are listed. Then, the decision to split the goal of estimation agent performance into time and accuracy estimation is explained. The GP domain for evolving time and accuracy estimation functions is proposed. For this purpose, the term *Type consistency* is introduced together with type consistent functions and terminals. Some of introduced functions and terminals utilize metadata and previous results. An architecture of an intelligent agent encapsulating the similar set identification and estimation functions is proposed.

4.1 Available metadata

Metadata values were extracted from all instances of all data mining tasks computed by our agents. Metadata is divided into two categories:

- Task related - all metadata that was available to each task:
  - Number of instances
  - Number of attributes
- Attribute related - all metadata that was available to each attribute of the task:
  - Type: determines the nature of the attribute and it has one of the following values – real, integer, categorical, Boolean. Additional availability of metadata depends on the type, as seen in Figure 4.1.

Type	Additional metadata
Real	Min, Max
Integer	Min, Max
Categorical	Number of categories
Boolean	

Figure 4.1 - Type dependant metadata

- Missing values: determines whether an attribute can have an unknown value or not.

Brief summary of metadata is shown in Figure 4.2.

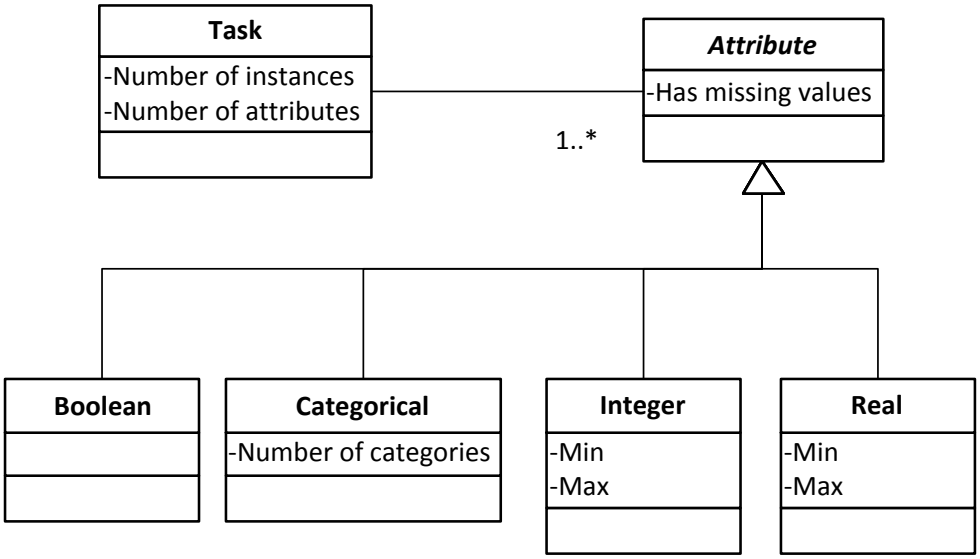


Figure 4.2 - Available metadata

4.2 Metrics between tasks

In our metric proposal we proceeded from the metric proposed in [8] (see 2.5.2 for a more detailed overview):

$$d(m_1, m_2) = \sum_{i=1}^{n} w_i \times d_i(m_1[i], m_2[i]),$$

where m_1, m_2 are metadata of compared tasks, w_i is a weight of each attribute and d_i is distance between metadata attribute i.

This metric yields different results for various orderings of metadata. Even the same data mining problems, only with different attribute ordering, may be evaluated as distant. We propose a metric that will eliminate this issue and that will consist of three parts:

$$d(m_1, m_2) = d_{categorical}(m_1, m_2) + d_{integer}(m_1, m_2) + d_{real}(m_1, m_2).$$

Several definitions will be stated before the exact proposal of these three distances. Also for the purposes of the metric, Boolean attributes will be treated as categorical attributes with two categories *True* and *False*. Let *min* be a minimum value of integer occuring in the metadata, likewise let *max* be a maximum value of integer occuring in the metadata. Let Σ be an alphabet of integers between *min* and *max.* A *string S* over an alphabet Σ is a (finite) concatenation of symbols from *S*. The *length* of a string *S* is the number of symbols in *S*, denoted by $|S|$. $S[i]$ denotes the i^{th} symbol of *S. Alignment* of two strings $S_1$and S_2 (without loss of generality, let us assume that $|S_1| \geq |S_2|$) is a function f that for every position in S_1 returns a position in S_2, or a special symbol *GAP.* In addition, for every position *j* in S_2, there must exist exactly one position *i* in S_1 such that $f(i) = j$.

Let the string representation of categorical attributes be a concatenation of the number of categories of categorical attributes. Let $C_2\,[GAP]$ be treated as 0. Let C_1, C_2 be a string representations of categorical attributes of metadata m_1, m_2. Then we define categorical distance as:

$$d_{categorical}(m_1, m_2) = \min_f \sum_{i=1}^{|C_1|} |C_1[i] - C_2\,[f(i)]|.$$

It is clear that the proposed categorical distance does not depend on the ordering of attributes. A simple algorithm for obtaining proposed canonical comparison can be derived directly from the definition by enumerating all the possible alignments of two data minining problems. Suppose that both problems have n categorical attributes. The complexity of the enumeration is then $O(n^n)$. We will show that the complexity of this simple algorithm can be significantly improved.

Observation: C_1, C_2 can be sorted in ascending order by the number of categories resulting in C_1', C_2'. A new alignment f' can be easily found that copies f in the sense of $\sum_{i=1}^{|C_1|} |C_1[i] - C_2\,[f(i)]|$. This sorted equivalent representation will be exclusively used.

We say that positions i, j are *inverted* if:

- $C_1[i] < C_1[j]$

- $f(i) \neq GAP$ AND $f(j) \neq GAP$
- $C_2\,[f(i)] > C_2\,[f(j)]$

Theorem 1: Let us suppose that positions *i,j* are inverted. Let alignment f' be defined as:

$$f'(x) = \begin{cases} f(j); & if\ x = i \\ f(i); & if\ x = j \\ f(x); & otherwise \end{cases}.$$

Then the number of inverted positions in f' is smaller than in f and:

$$\sum_{l=1}^{|C_1|} |C_1[l] - C_2\,[f(l)]| \geq \sum_{l=1}^{|C_1|} |C_1[l] - C_2\,[f'(l)]|.$$

Proof: Note that the two sums differs only in position *i,j*:

$$\sum_{l=1}^{|C_1|} |C_1[l] - C_2\,[f'(l)]| =$$

$$= \left(\sum_{l=1, l\neq i,j}^{|C_1|} |C_1[l] - C_2\,[f(l)]| \right) + |C_1[i] - C_2\,[f'(i)]| + |C_1[j] - C_2\,[f'(j)]|.$$

From 24 possible orderings of $C_1[i],\ C_1[j], C_2\,[f(i)], C_2\,[f(j)]$ only six of them are inverted. For the sake of simplicity, let us rename $C_1[i],\ C_1[j], C_2\,[f(i)], C_2\,[f(j)]$ to a, b, c, d (in the same order). For each possible order, the following condition needs to be verified:

$$|c - a| + |d - b| \geq |d - a| + |c - b|.$$

ORDER	$\|\boldsymbol{c} - \boldsymbol{a}\| + \|\boldsymbol{d} - \boldsymbol{b}\|$	$\|\boldsymbol{d} - \boldsymbol{a}\| + \|\boldsymbol{c} - \boldsymbol{b}\|$	$\geq$
ABDC	$\|d - c\| + 2 * \|d - b\| + \|b - a\|$	$\|d - c\| + 2 * \|d - b\| + \|b - a\|$	=
ADBC	$\|c - b\| + 2 * \|d - b\| + \|d - a\|$	$\|d - a\| + \|c - b\|$	$\geq$
ADCB	$2 * \|d - c\| + \|d - a\| + \|c - b\|$	$\|d - a\| + \|c - b\|$	$\geq$
DABC	$\|c - b\| + \|d - a\| + 2 * \|b - a\|$	$\|d - a\| + \|c - b\|$	$\geq$
DACB	$2 * \|c - a\| + \|d - a\| + \|c - b\|$	$\|d - a\| + \|c - b\|$	$\geq$
DCAB	$2 * \|c - a\| + \|d - c\| + \|b - a\|$	$2 * \|c - a\| + \|d - c\| + \|b - a\|$	=

The condition holds for every possible case.

The transformation from f to f' is depicted in Figure 4.3.

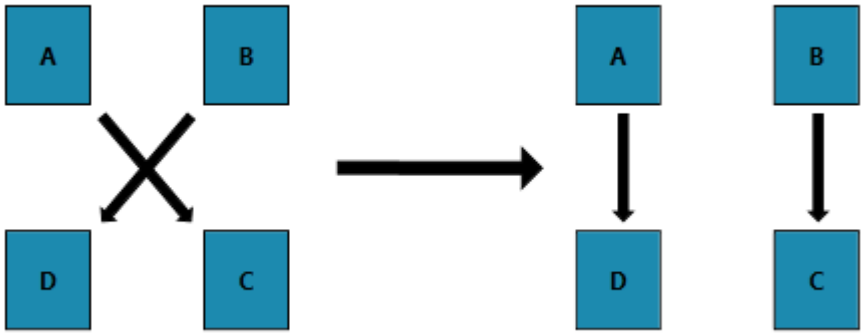
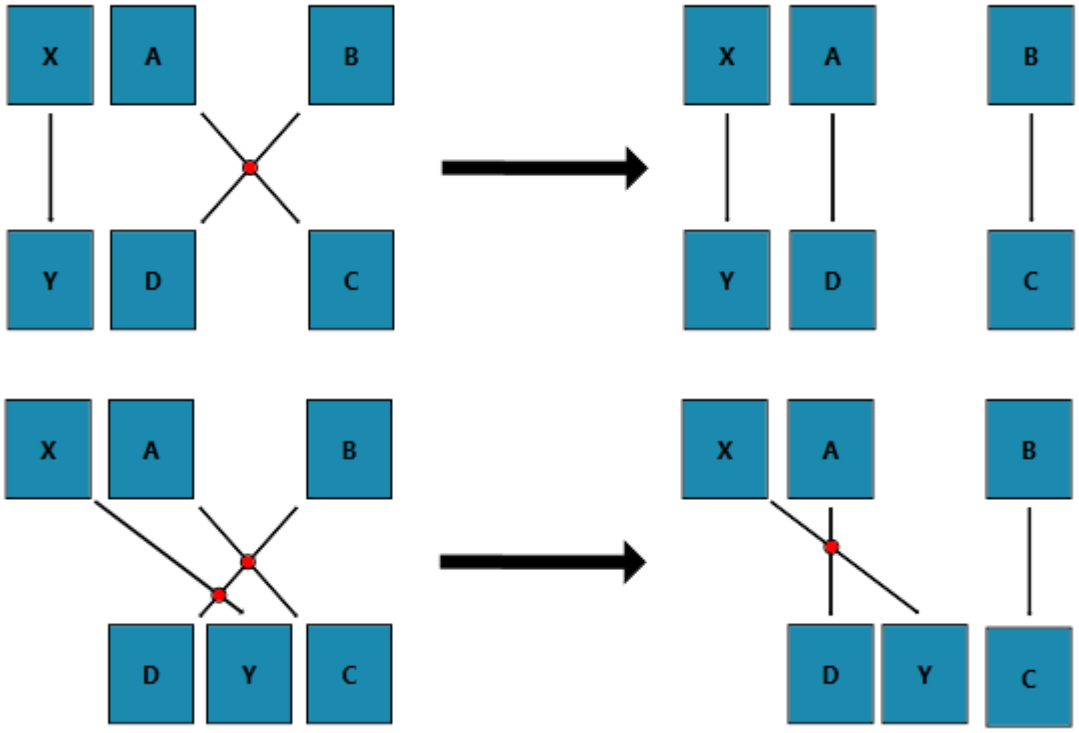
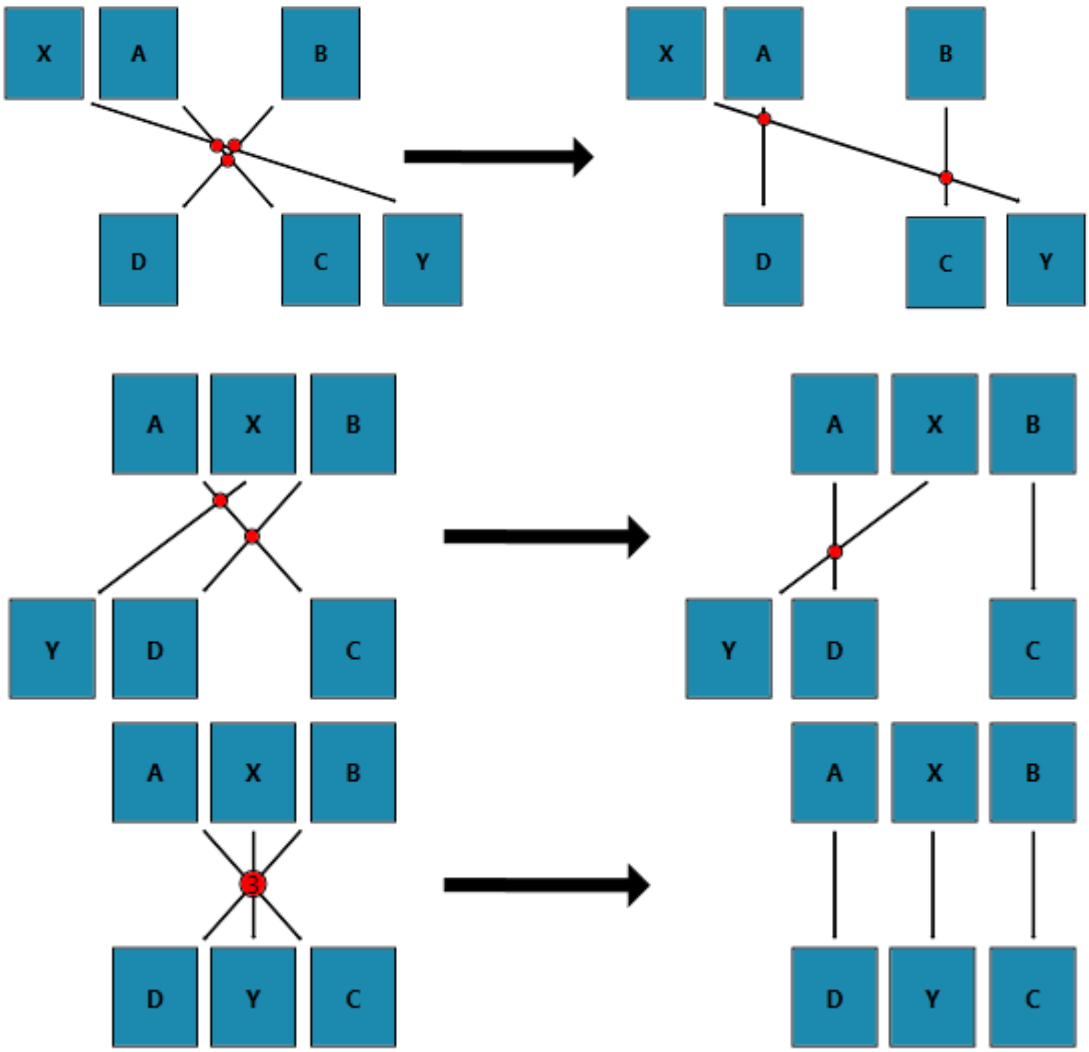


Figure 4.3 - Inverted positions transformation

Positions *i ,j* are no longer inverted. We will show that for each possible position *x* in C_1 and *y* in C_2, no new inverted positions appear. There are nine possible cases to be checked: 3 possible orders of *x* with relevance to *a*, *b* to each of 3 possible orders of *y* with relevance to *c*, *d*. 4 of nine positions are symmetric to other positions.





We have shown that no new inverted positions appeared and at least one inverted pair disappeared. The number of inverted positions in f' is smaller than in f.

Q.E.D.

Theorem 2: There exists a minimal alignment of C_1, C_2 with no inverted positions.

Proof: Suppose there is no optimal alignment without inverted positions. Take any optimal alignment. By repeated application of the previous theorem, new alignments are still optimal with decreasing number of inversions. Number of inversions is limited, thus, after a finite number of steps we finish with a minimal alignment without inversions.

Q.E.D.

This theorem improves the complexity of the algorithm computing the distance. Only alignments without inversions need to be enumerated. All we need is to sort the representations by the number of categories and then choose between the positions that project to *GAP*.

The best positions to project to *GAP* can be found by a Needleman-Wunsch algorithm [17] that is used in bioinformatics for finding an optimal alignment. Complexity of the Needleman-Wunsch algorithm is polynomial to the length of C_1, C_2. We will show that the complexity can still be improved.

Theorem 3: Alignment f defined as:

$$f(i) = \begin{cases} GAP; & if\ i \leq |C_1| - |C_2| \\ i - (|C_1| - |C_2|); & otherwise \end{cases}$$

is optimal.

Proof: Take any optimal alignment $f'(\neq f)$ with no inverted positions. We will show that it can be transformed to f without loss of optimality. In C_1 there must exist a position i such as:

$$f'(i+1) = GAP,$$
$$f'(i) = a \neq GAP.$$

By repeating the following transformation we will reach f:

$$f''(i+1) = a,$$
$$f''(i) = GAP.$$

The following condition needs to be verified (note that $C_1[i] \leq C_1[i+1]$):

$$|C_1[i] - C_2[a]| + C_1[i+1] \geq C_1[i] + |C_1[i+1] - C_2[a]|.$$

There are 3 possible orders of $C_1[i], C_1[i+1], C_2[a]$:

ORDER	$\|C_1[i] - C_2[a]\| + C_1[i+1]$	$C_1[i] + \|C_1[i+1] - C_2[a]\|$	$\geq$
$C_2[a], C_1[i], C_1[i+1]$	$C_1[i] - C_2[a] + C_1[i+1]$	$C_1[i] + C_1[i+1] - C_2[a]$	=
$C_1[i], C_2[a], C_1[i+1]$	$C_2[a] - C_1[i] + C_1[i+1]$	$C_1[i] + C_1[i+1] - C_2[a]$	$\geq$
$C_1[i], C_1[i+1], C_2[a]$	$C_2[a] - C_1[i] + C_1[i+1]$	$C_1[i] - C_1[i+1] + C_2[a]$	$\geq$

The condition holds for every possible order.

Q.E.D.

The example of the optimal alignment defined in the previous theorem is illustrated in Figure 4.4.

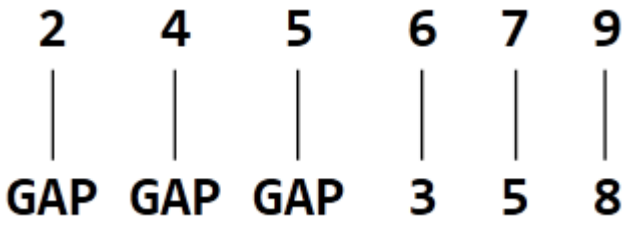


Figure 4.4 - Optimal alignment

By applying this theorem, the complexity of an algorithm computing the distance can be further improved. The algorithm is shown in Figure 4.5.

Categorical distance (C_1, C_2)

$C_1' \leftarrow$ sorted C_1

$C_2' \leftarrow$ sorted C_2

$Compute\ alignment\ f\ defined\ in$ Theorem 3

$$return \sum_{i=1}^{|C_1|} |C_1[i] - C_2[f(i)]|$$

Figure 4.5 - Categorical distance

The most time consuming part is now sorting of C_1, C_2 which has the complexity of $O\big(n * log(|C_1|)\big)$. Computing the alignment and summing up the distance has a complexity of $O(|C_1|)$.

Let us focus on integer distance now. Let the string representation of integer attributes be a concatenation of $|Max - Min|$ of integer attributes. Let I_1, I_2 be string representations of integer attributes of metadata m_1, m_2. We propose the integer distance in a similar way as the categorical distance:

$$d_{integer}(m_1, m_2) = \min_f \sum_{i=1}^{|C_1|} |I_1[i] - I_2[f(i)]|.$$

Again, the proposed distance does not depend on the order of attributes.

Observation: Theorem 1, Theorem 2 and Theorem 3 can be reformulated using I_1, I_2 instead of C_1, C_2.

Corollary: Algorithm for computing the categorical distance can be easily modified for computing the integer distance as well. The complexity of the algorithm remains unchanged -$O\big(n * log(|C_1|)\big)$.

The last part of the metric is the real distance d_{real}. Let *f*, *l* be optimal alignments for categorical and integer distance from Theorem 3. We propose d_{real} as follows:

$$d_{real}(m_1, m_2) = |R_1 - R_2|$$
$$* max(4, \{|C_1[i] - C_2[f(i)]||i \in dom(f)\}, \{|I_1[i] - I_2[l(i)]||i \in dom(l)\}),$$

where R_1, R_2 are number of real attributes in m_1, m_2. We have argued that real attributes do not depend on its Min and Max (because even if Min and Max are arbitrarily near, there is still infinity of different positions) and should be the most significant. Definition of the real distance does not depend on the order of real, integer and categorical attributes. It would be undesirable if there were other optimal f', l' alignments that would be more optimal in the sense of real distance compared to f, l. We will show that this is not the case.

Theorem 4: Any optimal alignment f', l' (where $f' \neq f$ and $l' \neq l$) can be transformed info *f*, *l* without an increase of elements in the maximum part of the real distance.

Proof: It should be clear that transformations used in Theorem 1 and Theorem 3 are sufficient to transform any optimal alignments into *f*, *l*.

We will show that transformation from Theorem 1 does not increase elements in the maximum part of the distance:

ORDER	$\boldsymbol{max}(\|\boldsymbol{c}-\boldsymbol{a}\|, \|\boldsymbol{d}-\boldsymbol{b}\|)$ **?** $\boldsymbol{max}(\|\boldsymbol{d}-\boldsymbol{a}\|, \|\boldsymbol{c}-\boldsymbol{b}\|)$
ABDC	$\|\boldsymbol{c}-\boldsymbol{a}\|$ (maximum distance in this order)
ADBC	$\|\boldsymbol{c}-\boldsymbol{a}\|$ (maximum distance in this order)

ORDER	$\boldsymbol{max(\lvert c-a\rvert, \lvert d-b\rvert)\ ?\ max(\lvert d-a\rvert, \lvert c-b\rvert)}$
ADCB	$\lvert c-a\rvert \geq \lvert d-a\rvert\ AND\ \lvert d-b\rvert \geq \lvert c-b\rvert$
DABC	$\lvert c-a\rvert \geq \lvert c-b\rvert\ AND\ \lvert d-b\rvert \geq \lvert d-a\rvert$
DACB	$\lvert d-b\rvert$ (maximum distance in this order)
DCAB	$\lvert d-b\rvert$ (maximum distance in this order)

We will now prove that transformation from Theorem 3 does not increase elements in the maximum part of the distance:

ORDER	$max(\lvert C_1[i]-C_2[a]\rvert, C_1[i+1])\ ?\ max(C_1[i], \lvert C_1[i+1]-C_2[a]\rvert)$
$\boldsymbol{C_2[a], C_1[i], C_1[i+1]}$	$C_1[i+1]$ (maximum element in this order)
$\boldsymbol{C_1[i], C_2[a], C_1[i+1]}$	$C_1[i+1]$ (maximum element in this order)
$\boldsymbol{C_1[i], C_1[i+1], C_2[a]}$	$C_1[i+1] \geq C_1[i]\ AND\ \lvert C_1[i]-C_2[a]\rvert \geq \lvert C_1[i+1]-C_2[a]\rvert$

Q.E.D.

This concludes our metric proposal and analysis.

4.3 Agent comparison

Now we will describe the data mining agent comparison. For every computational experiment the following results were available:

- Agent type: for example Multilayer perceptron.
- Options: agent's options (for example number of layers)
- Task processed: the task which was computed by the agent
- Error rate
- Kappa statistic
- Mean absolute error
- Root mean squared error
- Relative absolute error
- Root relative squared error
- Computation duration

Not all results were used in the algorithms proposal. We will state exact definitions when the results are used for the first time. For the exact definitions of all results, see [18].

Given a task, it is clear that out of two agents with the same accuracy, the agent able to compute the task faster is better. Likewise, given two agents with the same speed, the more accurate one is better. The problem occurs when comparing a fast agent with low accuracy and a slow agent with high accuracy. Which one is better? These multi-criteria decisions are always difficult because we sometimes need a fast agent and sometimes a more accurate one. Due to this we decided to make no a priori assumptions by separating the agent estimation into two different parts: time and accuracy, and let the inquiring agents (or user) aggregate these two parts, and compare the agents themselves. Inquirers may have more information about their current preferences. The option of creating the mechanism for obtaining these preferences was also discussed, but declined because of the possible complexity of those preferences. To propose a way of exchanging such preferences is a possibility for future work.

4.4 The GP proposal

Estimation functions from the previous chapter will be evolved by genetic programming (see 3.2) and will be represented by two trees: one for estimating accuracy and another one for estimating time consumption. Each estimation function will receive a task to estimate, task metadata, an agent to estimate, and performance results of the agent on similar tasks based on the metric proposed in 4.2 as inputs.

Because we have decided to evolve two trees (one for accuracy and one for time estimation) as explained in previous section, we need to propose two domains for the GP algorithm. Later in this section, functions and terminals will be proposed. If not said otherwise, the proposed functions and terminals can be used regardless of the type of program evolved. All functions and terminals will be type consistent, which means that all functions will have all arguments and output of the same type.

This type will be a real number in our case. If some n-ary function is not defined on the whole R^n, we will propose its extended definition on the whole R^n.

Functions

- Basic mathematical functions: add, subtract, multiply and divide will be used. Only division needs to be generalized. The following generalization was chosen:

$$function\ divide\ (a_1, a_2) = \begin{cases} \frac{a_1}{a_2};\ \ if\ a_2 \neq 0, \\ 0;\ \ otherwise. \end{cases}.$$

- Other functions: we have introduced the following mathematical functions into the domain: $square\ root\ and\ logarithm\ to\ base\ 2$. These functions were generalized by following:

$$function\ square\ root\ (a_1) = \begin{cases} \sqrt{a_1};\ \ if\ a_1\ \geq 0, \\ \sqrt{|a_1|};\ otherwise. \end{cases}$$

$$function\ logarithm\ to\ base\ 2\ (a_1) = \begin{cases} \log_2 a_1\ ;\ \ if\ a_1 > 0, \\ 0;\ \ if\ a_1 = 0, \\ \log_2 |a_1|;\ \ otherwise. \end{cases}$$

- Boolean functions: for the sake of type consistency all Boolean functions *b* with arity *i* were proposed according to the following pattern:

$$Boolean\ function\ b\ (a_1, .., a_i, a_{i+1}, a_{i+2})$$
$$= \begin{cases} a_{i+1};\ \ if\ b(a_1, .., a_i), \\ a_{i+2};\ \ otherwise. \end{cases}$$

  Namely:

$$Less\ than\ (a_1, a_2, a_3, a_4,) = \begin{cases} a_3;\ \ if\ a_1 < a_2, \\ a_4;\ \ otherwise. \end{cases}$$

$$Less\ than\ or\ equal\ (a_1, a_2, a_3, a_4,) = \begin{cases} a_3;\ \ if\ a_1 \leq a_2, \\ a_4;\ \ otherwise. \end{cases}$$

Some other functions were also discussed:

- Polynomial functions: We did not want to expand the domain too much, and this type of functions can be expressed by combining basic functions, so we did not introduce polynomials into population.

- Periodic functions: like *sin*, *cos*. We have argued that an evolving program will not benefit from periodicity, thus we did not introduce such functions into GP domain.
- Boolean functions *greater than*, *greater than or equal.* These were not introduced into the domain because they can be expressed by the means of Boolean functions already in the domain:

$$Greater\ than\ (a_1, a_2, a_3, a_4,) = Less\ than\ or\ equal\ (a_2, a_1, a_3, a_4,),$$
$$Greater\ than\ or\ equal\ (a_1, a_2, a_3, a_4,) = Less\ than\ (a_2, a_1, a_3, a_4,).$$

Terminals

- Constant terminals: we have proposed terminals that represent real and integer numbers. When creating such a terminal, a random number is generated and set as a value of the new terminal.
- Previous results terminals: terminals that represent previous results of the agent in question were proposed. When creating such a terminal a random number i is generated. Value of the new terminal is the root mean squared error of the agent on the ith nearest task for the accuracy domain and computation duration of the ith nearest task for the time domain.

  To measure time elapsed, the $Computation\ duration$ was the only option. Computation duration is the time interval between start and finish of the computation and it is expressed in seconds. For the accuracy results, we wanted to keep the domain increase to a minimum, so only one error characteristic was chosen. The $Root\ mean\ squared\ error$ (RMSE) is widely used, so we have decided to use it as well. RMSE of the estimator Φ' to the estimated parameter Φ is defined as the square root of the mean square error: $RMSE(\Phi') = \sqrt{E((\Phi' - \Phi)^2)}$.

  If the agent computed the $i-th$ nearest task multiple times (with modified input parameters), it was not clear which results to choose. We have argued that we want to estimate the accuracy potentials of the agents, and it is up to the researcher to find the best settings, thus choosing the results with the

best accuracy for the accuracy domain. It would be logical to apply the same for the time domain, but as it turned out, many algorithms had zero computation duration expressed in seconds with certain settings. Therefore we have decided to choose average computation duration for the time domain, which can be also viewed as an average time to test particular settings for some agent.

- Metadata terminals: We could propose a terminal for each type of metadata available, but we argued that this amount of terminals is not necessary to successfully estimate time or accuracy. Instead we proposed the following terminal that aggregates metadata information of the task on input:

$$terminal\ complexity = \sum_{a_i} \delta(a_i) \times \log_2(Task.NumberOfInstances),$$

where a_i is an i-th attribute of the task, $\delta(a_i)$ is the complexity of an attribute defined as:

$$\delta(a_i) = \begin{cases} 1; & if\ a_i\ is\ boolean\ or\ categorical\ attribute \\ 2; & if\ a_i\ is\ integer\ attribute \\ 3; & if\ a_i\ is\ real\ attribute \end{cases}$$

and $Task.NumberOfInstances$ is the number of instances of the task on input.

The purpose of this terminal is to represent task complexity. We have argued that integer attributes are nearly always harder to compute than Boolean and categorical attributes, and similarly real attributes are nearly always harder to compute than integer attributes. The complexity of the task also rises with a logarithm of the number of instances.

Random number terminals were also discussed. We have argued that an evolved program would not benefit from stochasticity, therefore we did not introduce such terminals into domain. Note that this is different from generating constants, because random number terminals generate a new number each time they are evaluated.

4.5 Estimation agent architecture

The Estimation agent architecture is proposed in this section. Suppose that following things are available:

- Evolved GP trees (for accuracy and time estimation) accepting task complexity and previous results. Functions represented by trees return performance estimation.
- Task metadata. The abstract place where the metadata is stored will be called *Metadata storage*. The exact form of this storage is implementation dependent.
- Performance results of previous computations. The abstract place where the results are stored will be called *Computation results storage*. Again, the exact form of this storage is implementation dependent.

We suppose that all agents save their performance results into the Computation results storage and that all tasks have their metadata saved in the Metadata storage.

Our agent consists of 4 parts – the *input* part, the *zooming* part, the *estimation* part and the *output* part. The input part is listening for queries and forwards them to the zooming part. The zooming part handles zooming and forwards the results to the estimation part. Another role of the zooming part is to obtain the data from both storages. The estimation part encapsulates estimation trees. Its goal is to return a time and accuracy estimation of all agents that occur in Computation results storage on the task in the query. The output part handles the estimation results delivery to the inquirer. All parts and their purposes are overviewed in Figure 4.6.

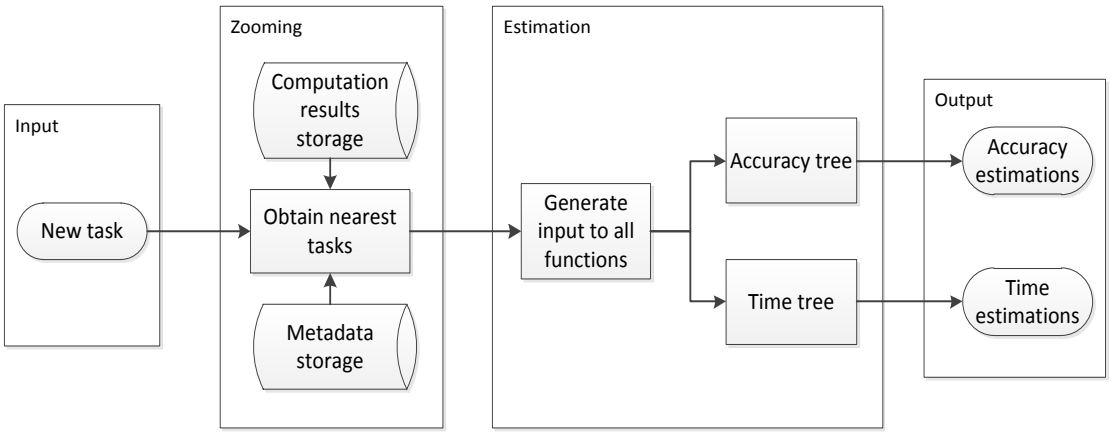


Figure 4.6 - Architecture overview

For each agent in the Computation results storage, the estimation is as following:

- In zooming, we first identify several closest tasks already computed by the agent based on their metadata and metrics proposed in 4.2. For this process, we use the actual states of Computation results and metadata storage. The number of tasks obtained by zooming depends on the GP trees evolved, to be more precise, on the highest argument of the Result terminal used in the trees.
- In estimation, the input task and the tasks obtained by zooming are used to compute an input for all terminals in the generated trees. The trees are evaluated and their output is returned.

The output part saves the estimation of each agent to an array. The array is then forwarded to the inquiring agent.

In chapter 2, we stated that our goal is a design of an intelligent agent. Let us see if the proposed architecture satisfies all three conditions:

- reactivity: The agent listens to new queries and computes them immediately.
- pro-activeness: New performance results are stored into the Computation results storage and new task metadata are available in Metadata storage. During zooming, the actual states of both storages are used.

- social ability: The agent is listening for queries and returns the result if the query is received.

5 Implementation

5.1 JADE

Jade stands for Java Agent DEvelopment Framework, and according to [19], it is an enabling technology, middleware for development and runtime execution of peer-to-peer applications which are based on the agent paradigm and which can seamlessly work and interoperate both in wired and wireless environment. The environment can evolve dynamically with agents that appear and disappear in the system according to the needs and requirements of the application environment. Communication between the agents is completely symmetric with each agent being able to play both the initiator and the responder role. JADE is fully developed in Java and is fully compliant with the FIPA (the Foundation for Intelligent Physical Agents, an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies) specifications. As a consequence, JADE agents can interoperate with other agents, provided that they comply with the same standard. JADE provides a homogeneous set of APIs that are independent from the underlying network and the Java version. Programmers do not need to use all the features provided by the middleware. Features that are neither used do not require programmers to know anything about them, nor add any computational overhead.

JADE includes both the libraries (i.e. the Java classes) required to develop application agents and the run-time environment that provides the basic services and that must be active on the device before agents can be executed. Each instance of the JADE run-time is called *container* (since it "contains" agents). The set of all containers is called *platform* and provides a homogeneous layer that hides to agents (as well as to application developers) the complexity and diversity of the underlying tiers (hardware, operating systems, types of network, JVM) [19].

From the functional point of view, JADE provides the basic services necessary for distributed peer-to-peer applications in the fixed and mobile environment. JADE allows each agent to dynamically discover other agents and to communicate with them according to the peer-to-peer paradigm. From the application point of view,

each agent is identified by a unique name and provides a set of services. It can register and modify its services and/or search for agents providing given services, it can control its life cycle and, in particular, communicate with all other peers. Agents communicate by exchanging asynchronous messages, a communication model almost universally accepted for distributed and loosely coupled communications. In order to communicate, an agent sends a message to a destination. Agents are identified by a name and, as a consequence, there is no temporal dependency between communicating agents. The sender and the receiver need not be available at the same moment. The receiver may not even exist (or not yet exist) or may not be directly known by the sender that can specify a property. The structure of a message complies with the *Agent Communication Language* defined by FIPA and modelling speech act theory (see 2.2). A message includes fields, such as variables indicating the context a message refers-to and the timeout that can be waited for before an answer is received, aimed at supporting complex interactions and multiple parallel conversations. To facilitate the creation and handling of the content of messages, JADE provides support for automatically converting back and forth between the format suitable for content exchange, including XML and RDF, and the format suitable for content manipulation (i.e. Java objects). This support is integrated with some ontology creation tools, e.g. Protégé, allowing programmers to graphically create their ontology. JADE is opaque to the underlying inference engine system, if inferences are needed for a specific application, and it allows programmers to reuse their preferred system. To increase scalability and to meet the constraints of environments with limited resources, JADE provides the opportunity of executing multiple parallel tasks within the same Java thread. Several elementary tasks, such as communication, may then be combined to form more complex tasks structured as concurrent Finite States Machines. In the J2SE and Personal Java environments, JADE supports mobility of code and of execution state. That is, an agent can stop running on a host, migrate onto a different remote host (without the need to have the agent code already installed on that host), and restart its execution from the point it was interrupted. This functionality allows, for example, distributing computational load at runtime by moving agents to less loaded machines without any impact on the application. The platform also includes

a naming service (ensuring each agent has a unique name) and a yellow pages service that can be distributed across multiple hosts [19].

5.2 Metadata extraction

A custom tool was implemented for the metadata (see 4.1) extraction. The tool searches for the files with *.data* extension. Each file represents a different data mining task. Each line of the file represents one instance of the task. Values for each instance are separated by commas. The tool extracts all metadata from these instances. Extracted metadata are then saved to an *.xml* file. XML schema copies the hierarchy described in 4.1 and depicted in the Figure 4.2.

5.3 Genetic programming framework

An overview of genetic programming can be found in 3.2.

For the purposes of GP experiments in this thesis, a GP framework was created. The design of the framework is based on two observations:

- Genetic programming experiments are often computationally expensive. With this in mind we wanted to allow parallel computation with an ability to add a computer to the computing cluster at any time.
- In previous experiments with genetic programming we have learned that the most time consuming part of the genetic programming algorithm is the fitness evaluation of the entire population; other parts of the algorithm were computed comparatively very quickly.

The client-server architecture seemed to be the most suitable for our GP implementation. The server stores the population and executes genetic operators. Clients, on the other hand evaluate individuals. The server is a web service that provides individuals that have not been rated yet to the clients. Clients evaluate them and return the results back to the server. When all individuals are evaluated, genetic operators are executed by the server and a new generation is created.

5.3.1 Server

The server was implemented in .NET using Windows Communication Foundation (WCF). WCF is a runtime and a set of APIs (application programming interface) in the .NET Framework for building connected, service-oriented applications. WCF can be easily configured to create only one service instance at time (which is ideal for our purposes where we need only one GP instance storing all data). The key features of the server are:

- Data storage: The server stores all individuals, function, terminals, genetic operators and historical data (for example, the fitness of earlier generations).
- Data exposure: The server provides an interface through which information about stored data can be obtained.
- Different client support: The server acts as a W3C standards compliant web service and exposes information about itself in a WSDL format. Almost all platforms have the ability to automatically create classes (or files) necessary to consume the service by processing the WSDL file.
- Parallel evaluation support: The server supports parallel evaluation of individuals. It stores a list of non-rated individuals. These are sent to clients for evaluation. An individual that has not been rated yet is not sent for the second time if there is an individual who has not been sent. If a client returns an evaluation result, the evaluated individual is removed from the list until it is altered by some genetic operator (so that the same individuals are not evaluated over and over again).
- Problem independence: The server has no idea about how the fitness is computed, so it can be immediately reused for a new problem (only a new set of functions and terminals may be necessary to set).
- Automatic serialization of the experiment state: The server, at the beginning of a new generation, automatically serializes and saves the experiment state (population, individuals, functions and so on). Saved files can be loaded in the future and the experiment can continue (even with altered genetic operators)

- Support of multi-population experiments: The server supports concurrent populations. A switch to a next generation occurs after all individuals from all populations have been evaluated. Populations may (but do not have to) cooperate.
- Support of global operators: The server supports genetic operators that alter more than one population.
- Support of complex genetic operators: Genetic operators can be easily modified to change the execution order of operators; operators can be also executed multiple times per next generation step (for example an implementation of the elitism operators must be executed twice – once for saving the best individuals and the second time for reinserting those two individuals into population). Operators can also reserve some places in the population and pass custom information to upcoming operators. In a multiple population experiment, each population can possess its own set of genetic operators.

5.3.2 Client

Every programming language capable of consuming standard web services can be used for client implementation. In our case the client was implemented in C#. The user interface was designed with Windows Presentation Foundation (WPF) using the Model View ViewModel (MVVM) architectural pattern. MVVM clearly separates the logic (model) and the graphical user interface (view). The view model acts as a value converter converting data from the model to the user interface. For the implementation of the fitness function specified in 4.4, the SQL server was used for storage of performance results. Tasks with metadata were stored in XML file. The key features of the client are:

- Evaluation of individuals: The client requests a not yet rated individual from the server, evaluates it (computing fitness, number of nodes, width and depth), and returns the results back to server.
- Individual visualization: The client is able to visualize individuals. Inner nodes and leaves are distinguished by color (green for inner nodes and blue for leaves). Nodes are labeled by the value of the operator/operand used. All

information relevant to the individual is also shown. An example of the visualization is depicted in Figure 5.1.

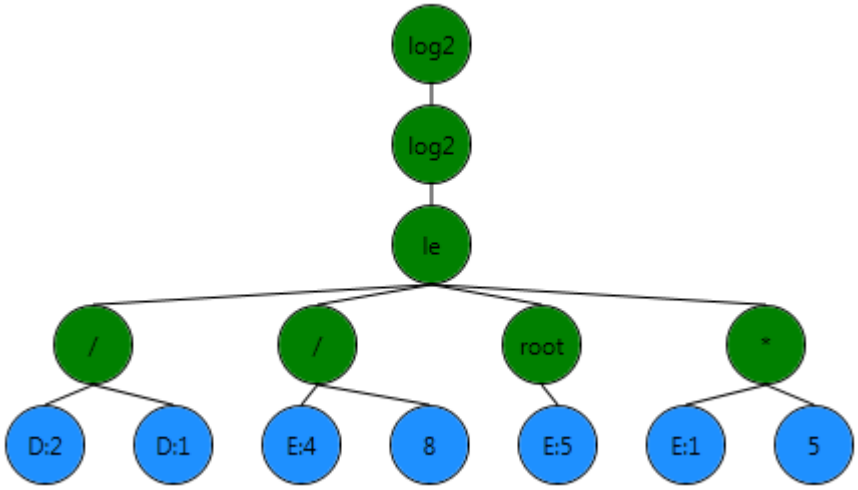


Figure 5.1 - Visualisation of an individual

- Progress visualization: The client is able to visualize the population's best fitness, average fitness, fitness distribution, average number of nodes, average width and depth of an individual of each population or of all populations combined.
- Saving of the best individuals: The client automatically saves visualization of the best individual (if the best individual is found during the evaluation phase) to a file. The name of the file contains information about the generation number and fitness.
- Progress export: The client can export the data about generation progress, specifically the best fitness, average fitness, average number of nodes, average width and depth of an individual of each generation, and fitness distribution of the last generation.
- Controlling the server: The client is able to reinitialize the server with one of the predefined initializators, and to load a specific, previously saved computation progress on the server.
- Task explorer: The list of all tasks can be viewed together with the distance between them and the available metadata of those tasks.

- Server connection verification: The client periodically checks whether a connection with server can be established. User is informed about the result of the last check.

5.3.3 Evaluation architecture

The client always initializes communication with a server. When the model is initialized (application start), it loads all tasks from the XML file, calculates the distance matrix of tasks, retrieves performance results from the database and forms a training set by combining distance matrix with performance results. After the user clicks in the GUI on a button initializing the evaluation, the view contacts model to start the evaluation. A background thread is created which evaluates the individuals until there is an update from the view telling the model to stop the background thread. To evaluate individuals, the following steps are repeated:

- A WCF client is created by background thread. The client acts as a proxy between the model and the server. The proxy communicates with the server using the HTTP protocol. Objects transferred across the network are serialized using Simple Object Access Protocol (SOAP) before the transmission, and deserialized again by using SOAP after the transmission. The proxy formulates the request for non-rated individual and sends it to server.
- The request is accepted by an IIS server and deserialized by WCF. The list of non-rated individuals is checked. If the list is empty, the server proceeds to the next generation by applying the operators. The list is refilled in the process. An individual from the list is serialized and returned back to proxy.
- The proxy deserializes the accepted individual. Fitness of this individual is calculated by using the training set. The proxy serializes the result together with an individual identification and sends it back to the server.
- The server stores the fitness of the individual and removes the individual from the list of non/rated individuals.

5.4 Estimation agent

The estimation agent implements the Estimation agent architecture (see 4.5) using JADE (see 5.1). JADE was chosen because of its standards compliance, wide usage and multi-platform support. If there are no queries for the Estimation agent, its thread is sleeping. When the query is received, the agent computes estimation of all agents as described in the architecture. *Metadata storage* is an XML file, *Computation results storage* is a database located on the Microsoft SQL Server 2008. The agent is able to load estimation trees from the server of the genetic programming platform (see 5.3.1). The loaded individual can be serialized and persisted between runs.

6 Experiments

This chapter describes experiments performed for evolving estimation trees (see 4.4) including the pre-processing phase. Results obtained are evaluated and discussed. Problems encountered during experiments are also mentioned.

We had 109732 records of previous computation results at our disposal for training our models. Previous experiments focused on trying different settings for few agents, thus the amount of distinct pairs (agent, task) was not as high as expected. Problems computed were commonly known problems from [20] and [18]. Calculated distance matrix for some selected tasks using metric proposed in 4.2 is shown in Figure 6.1.

	Breast-w	**Car**	**Haberman**	**Iris**	**Letter-recognition**
Breast-w	0	104	143	118	183
Car	104	0	139	38	283
Habermann	143	139	0	329	298
Iris	118	38	329	0	355
Letter-recognition	183	283	298	355	0
Lung-cancer	144	122	203	118	295
Machine	100380	100454	100327	356186	100419
Tic-tac-toe	108	12	143	42	289
Weather	102	70	69	180	259
Wine	16863	16938	16240	11308	16954
	Lung-cancer	**Machine**	**Tic-tac-toe**	**Weather**	**Wine**
Breast-w	144	100380	108	102	16863
Car	122	100454	12	70	16938
Habermann	203	100327	143	69	16240
Iris	118	356186	42	180	11308

	Lung-cancer	Machine	Tic-tac-toe	Weather	Wine
Letter-recognition	295	100419	289	259	16954
Lung-cancer	0	100506	128	146	16973
Machine	100506	0	100462	100390	786822
Tic-tac-toe	128	100462	0	74	16942
Weather	146	100390	74	0	16527
Wine	16973	786822	16942	16527	0

Figure 6.1 - Distance matrix

In 109732 rows of performance results, there were 43 unique pairs of (agent, task). We have observed that GP evolving accuracy estimation trees have always poor performance on some pairs. The histogram of the best root mean squared error of each pair, depicted in Figure 6.2, shows possible reason for this poor performance. Seven pairs have their results very far from the rest of the group (especially 4 pairs with RMSE above one billion indicated faulty records; which was proved by further investigation). In addition, the 3 remaining distant pairs did not share the same agent, thus the estimation of the one remaining pair could not be based on the previous computation of the other two pairs. The training set for accuracy estimation was therefore modified not to contain those 7 pairs. The size of the resulting training set was thus 36 pairs.

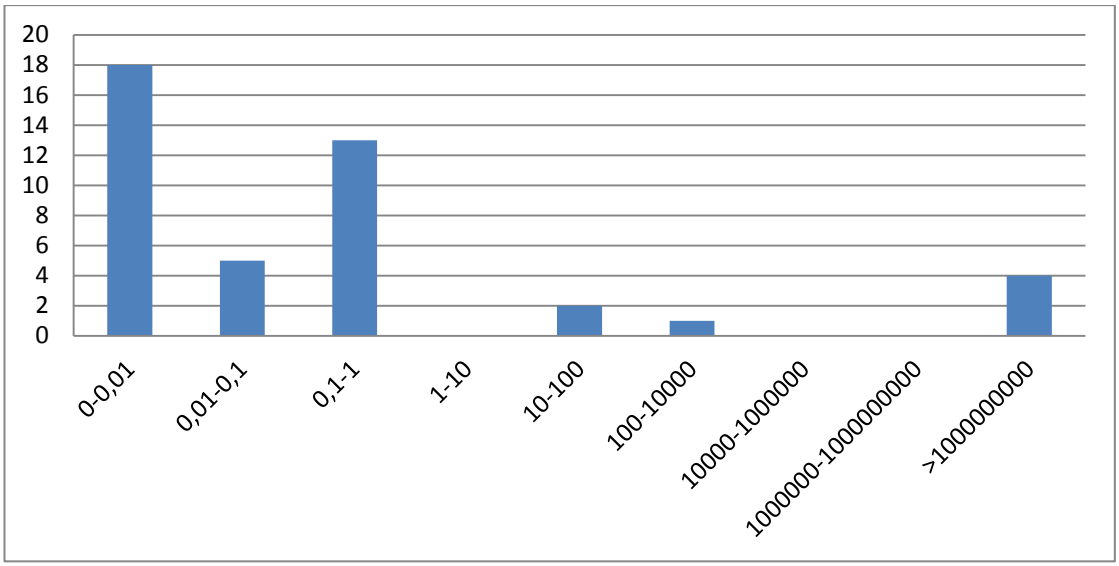


Figure 6.2 - RMSE histogram

For the time domain we used all 39 non faulty pairs. Histogram of computation duration expressed in seconds is shown in Figure 6.3. There are also some pairs quite distant from the rest of the population. Those pairs shared an agent, so we did not remove them from the training set, because we hoped that previous results terminals have the potential to deal with this problem.

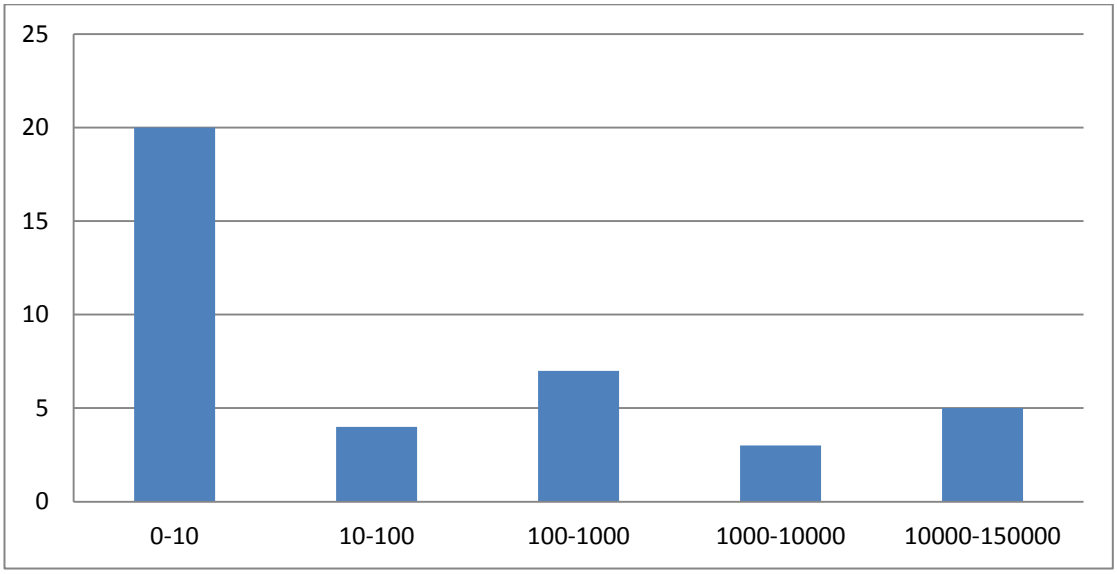


Figure 6.3 - Duration histogram

6.1 BLOAT

At the beginning of our experiments we have encountered bloat (for a definition of bloat see 3.2.5). To examine the encountered bloat, let us look at the generation

progress of one of our early experiments, ilustrated by Figure 6.4, and concentrate on the best fitness first. As one can see, there are larger fitness impovements in generations number 48, 58 and 66. After that the fitness of the best individual is rising very slowly (the best fitness difference of generation 67 and 156 is only about 0,37 and there is no significant increase between the two consequent generations). Since the last increase (generation 67) the average number of nodes, average width, and average depth of individual is beginning to grow. Note that the number of nodes can be exponential to the depth and width (thus explaining different rate of growth between number of nodes and depth/width).

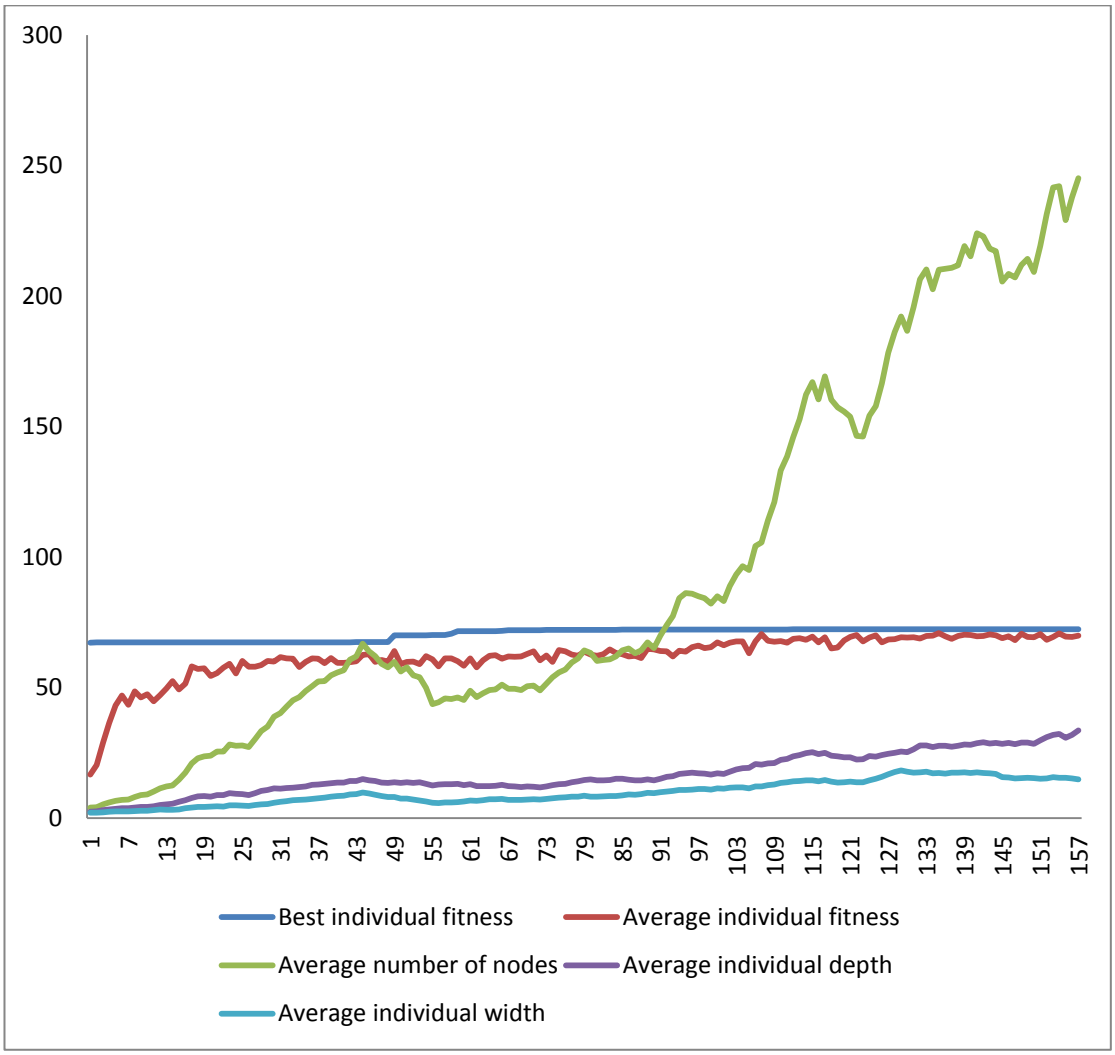


Figure 6.4 - Generation progress

The rate of bloating is illustrated by Figure 6.5 where visualized shapes of best individuals of selected generations can be seen.

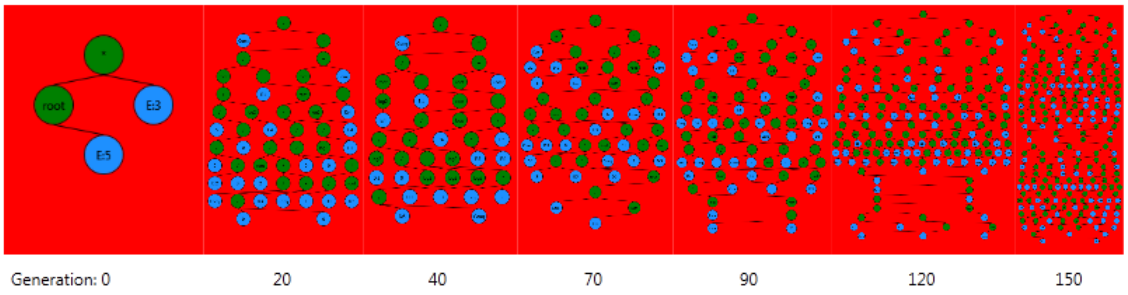


Figure 6.5 - Best individual progress

To apply correct countermeasures against the bloat we have examined to the detail one of the earliest generations in which bloat could be observed (generation 90). Fitness distribution of this generation is shown in Figure 6.6.

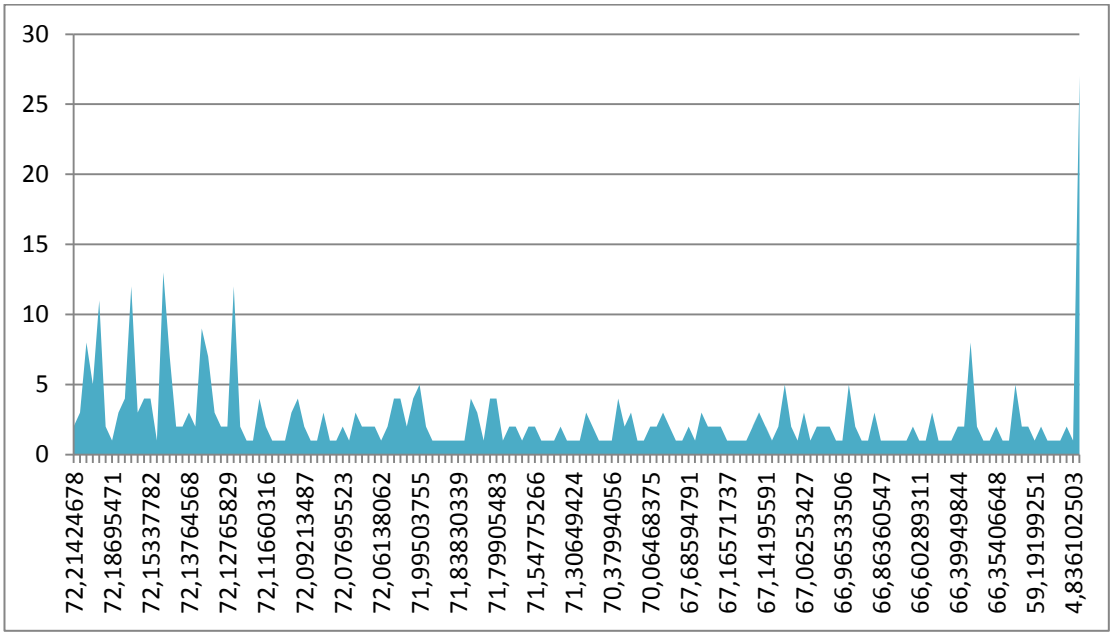


Figure 6.6 - Fitness distribution

At first glance, it could seem that the diversity of the population is good – average quantity of individual in population is below 5. On the other hand, almost all individuals have nearly the same fitness, which is unproductive, because the evolution cannot distinguish very well among individuals. A detailed look at the individuals also revealed that almost all individuals share the same upper part of the

tree, thus the evolution was only altering the lower parts of the tree. Therefore the diversity is lower than expected. To prevent this kind of bloating we decreased the selection pressure (the probability that a better individual is selected instead of worse individual), increased the population diversity (this was achieved by increasing number of populations, by increasing the interval after which individuals between populations are exchanged and by increasing the population sizes). We also used a slightly modified version of the anti-bloat selection (see 3.2.5) where we decreased the fitness of individuals that exceeded some size limits. The decrease was exponentially proportional to the amount that exceeded the limit.

6.2 Performance estimation experiments

Our main experimental work focused on modelling agents performance estimation. Experiments were performed with 4000 individuals in one generation. Individuals were divided into 10 populations. Every 10^{th} generation, the best individuals were exchanged between populations. Evaluation of 200 generations took approximately two hours on the Intel I5 processor. About 3 GBs of memory were needed to serialize 200 generations. Ramped half initialization (see 3.2.1) and Tournament selection (see 3.1.1) were used together with standard tree mutation and crossover (see 3.2.2 and 3.2.3). Elitism (see 3.1.1) and antibloat operator (see 3.2.5) scaling the fitness of large individuals were also used. All functions and terminals described in 4.4 were used. Constant terminals were generating integers between 1 and 10, previous results terminals value was in range between 1 and 5.

Individuals are visualized as described in 5.3.2. Terminals and functions used are listed in 4.4 with the following abbreviations in Figure 6.7:

Node	Value	Node	Value
	Distance of the first nearest task		Root square function
	Previous result of the first nearest task		Base 2 logarithm function

Node	Value	Node	Value
	Integer terminal 0		Addtition function
	Complexity of the task at hand		Substraction function
	Multiply function		Divide function
	Less than or equal		Less than function

Figure 6.7 - Terminal and function set abbreviations

6.2.1 Accuracy estimation experiments

This section describes accuracy estimation experiments. Because of the range of root mean squared error in training set and values of constant terminals, all RMSE were scaled by multiplication by 10. The fitness was calculated by substracting the error on the training set from 1000. Even some of the individuals in the first generation had fairly good fitness compared to the best individuals found. Example of a good individual from the first generation is shown in Figure 6.8. The fitness of this individual is 972,658082509677.

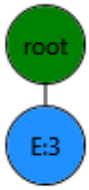


Figure 6.8 - Good initial individual

A wide range of different operator settings was tested. The best results were achieved with the following setting:

- Intialization: Depth of generated individuals up to 5.
- Tournament selection: Probability of victory of the best individual in the tournament 0.55 (very low selection pressure), tournament size 3.
- Mutation: Chance of an individual to be mutated 0.05, new individual generated by the grow method, depth of new individual up to 5.

- Crossover: Chance of an individual to be crossed 0.4.

One of the fittest inviduals is depicted in Figure 6.9. From our point of view this was the individual with the best ratio of fitness and size. The reason for this was that smaller individuals often have better ability to generalize. The fitness of this individual was 988,782426956407. This individual was discovered in generation number 158. Average error on the training set was approximately 0.31. One of the advantages of this individual is that at the top level of the tree, many *lesser than* functions can be found. Therefore, the individual has adapted to different scenarios.

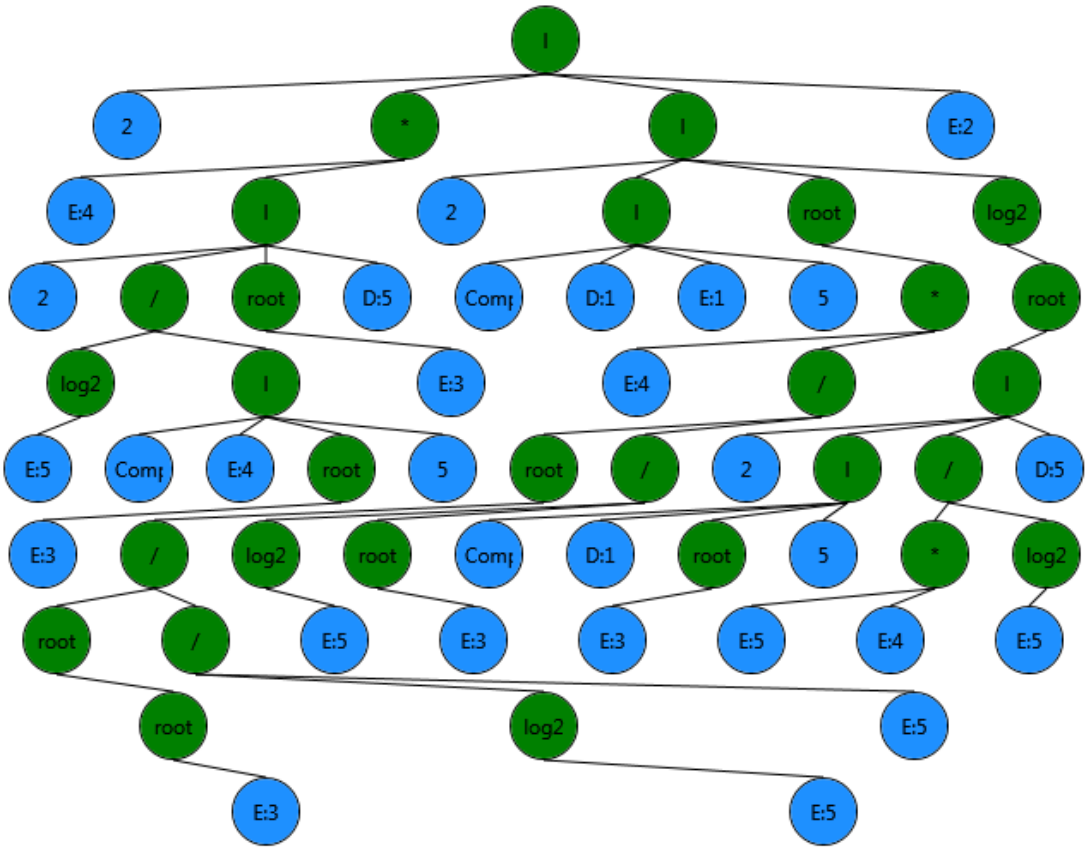


Figure 6.9 - The individual for accuracy estimation with the best ratio of fitness and size

The computation progress of the experiment in which the individual was obtained is shown in Figure 6.10. It is clear from the graph that there were lots of best individual improvements during computation, some of them were quite significant.

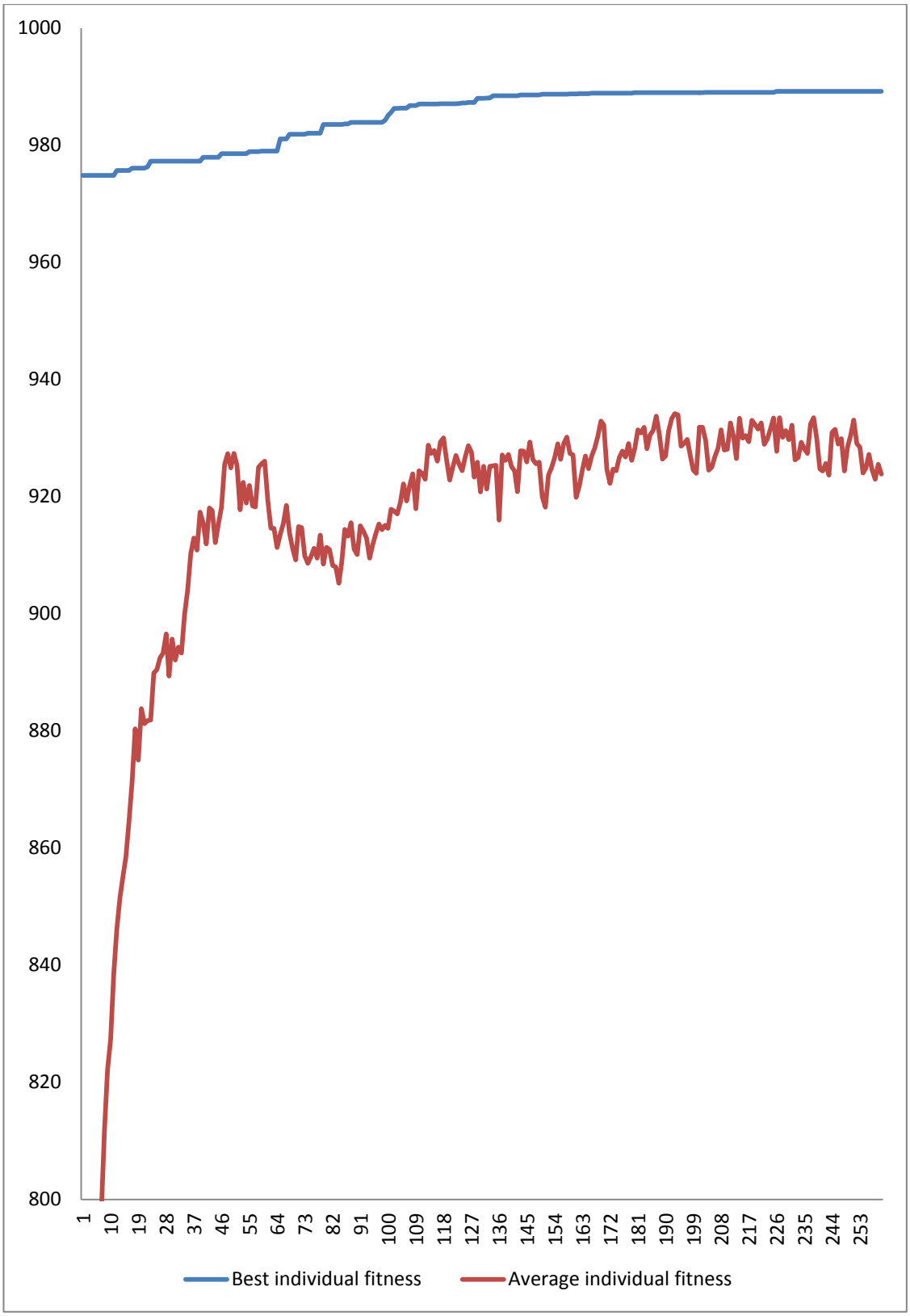


Figure 6.10 - Accuracy estimation computation progress

6.2.2 Time estimation experiments

This section describes the time estimation experiments. Compared to accuracy estimation experiments, the individuals in the first generation had poor performance compared to the best individuals found. We wanted all individuals to have nonnegative fitness. This forced us to scale the fitness used in accuracy estimation experiments by subtracting the error on the training set from 100 000. Experiments showed that individuals were wider than the individuals from accuracy estimation experiments because of the frequent use of Boolean functions with high branching factor. This could be changed by increased mutation frequency but with no significant effect on fitness. Some individuals have the tendency to distinguish two cases. One Boolean function identified the cases with low duration, and the corresponding subtree returned some fixed small number (resulting in small error). The rest of the tree computed cases with high duration. Example of such an individual is depicted in Figure 6.11. This individual returns 4 if it identifies low duration case. This individual's fitness was quite high: 9817929,45351489. Similar individuals had often high fitness values making this strategy popular during computation.

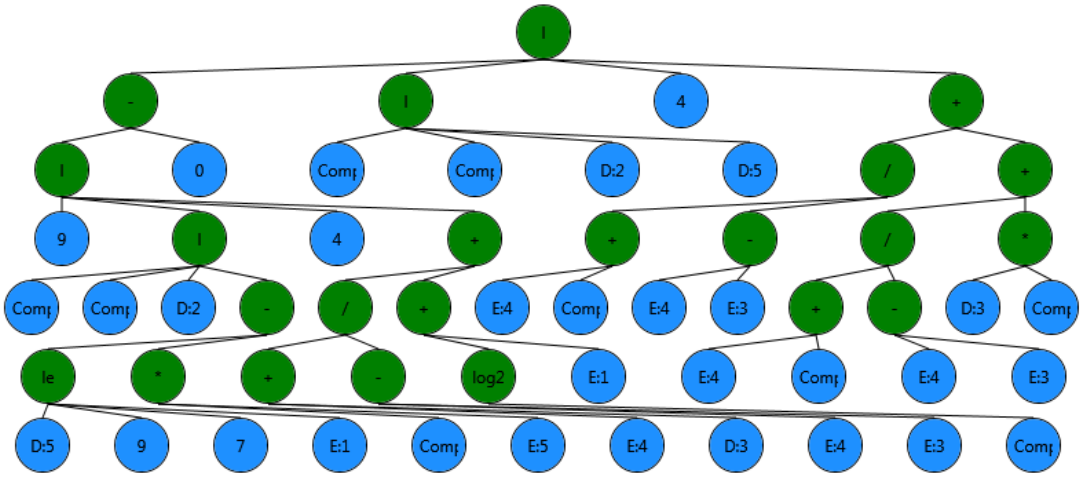


Figure 6.11 - Cunning individual

The best individual found computes both the above mentioned cases. It is shown in Figure 6.12. It was dicovered in generation 104. Fitness of the best individual

is 973472,021833934. The average estimation deviation from the test cases was about ten percent of test cases duration. Settings of the best run were as follows:

- Initialization: Depth of generated individuals up to 5.
- Tournament selection: Probability of the victory of the best invidual in the tournament 0.7, tournament size 3.
- Mutation: Chance of an individual to be mutated 0.05, a new individual generated by the grow method, the depth of new individual up to 5.
- Crossover: Chance of an individual to be crossed 0.4.

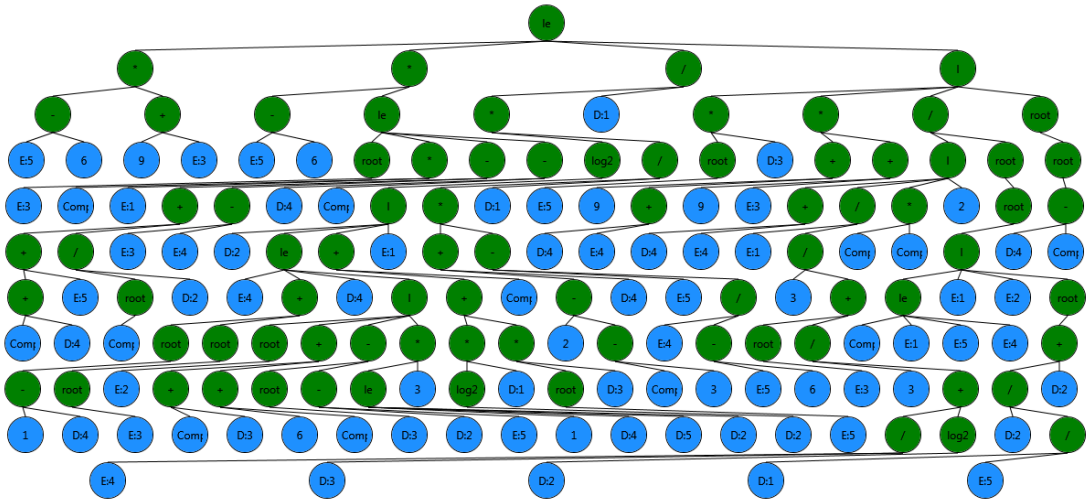


Figure 6.12 - The best duration estimation individual

Computation progress of the best run is shown in Figure 6.13. The fitness increase was even more rapid compared to accuracy estimation experiments.

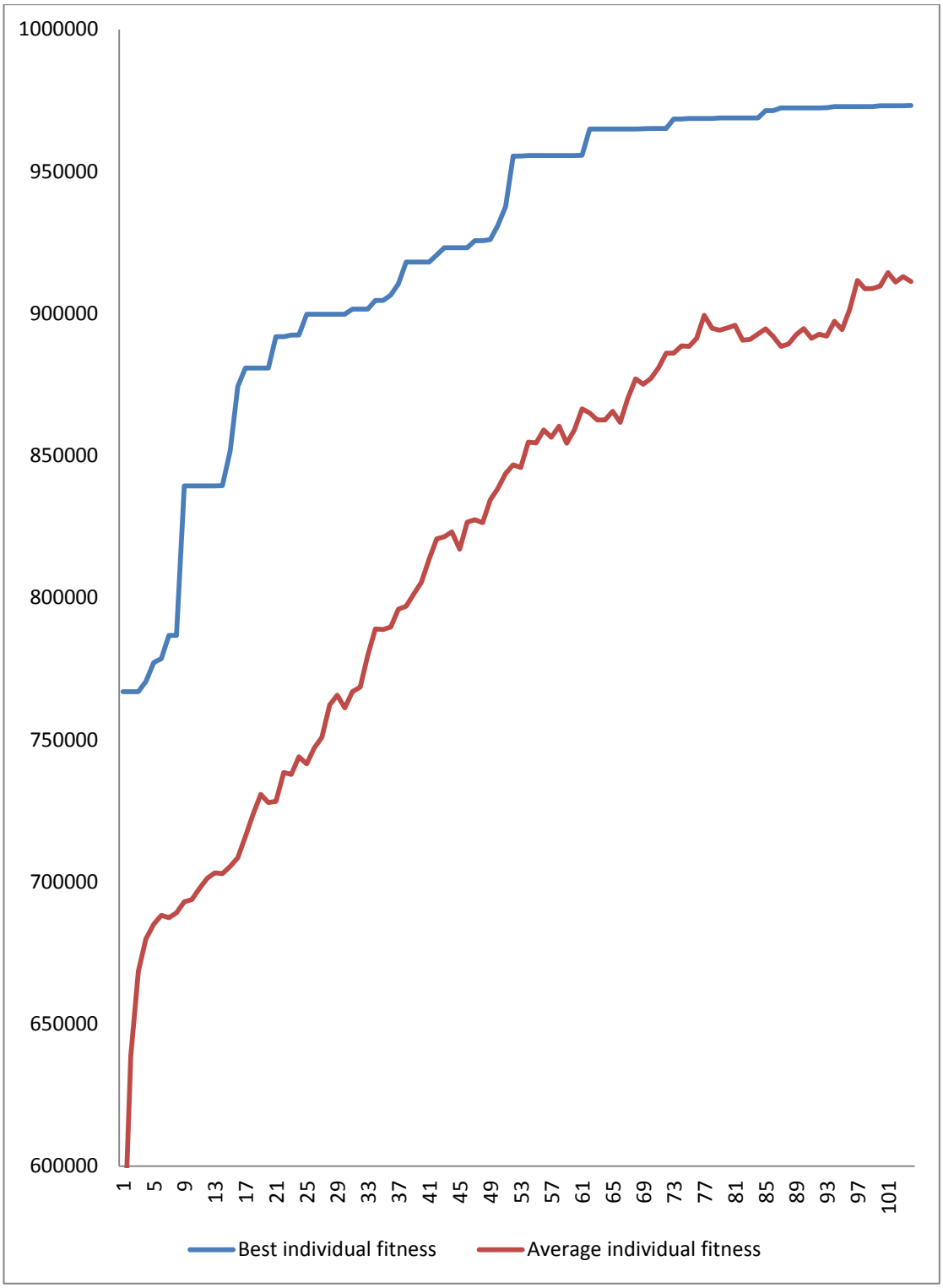


Figure 6.13 - Time estimation computation progress

6.3 Results validation

This section describes estimation performance of the two representative individuals obtained in the previous section on the validation set. The validation set was extracted from 649964 previously unseen performance records. These records were

not cleaned, so faulty data or misleading results could occur. There were 60 unique pairs of (agent, task). We have used these pairs as the validation set for both accuracy and estimation representative individuals validation.

6.3.1 Accuracy validation

Accuracy results of the 60 pairs were between 0 and 1 and that interval was well covered. Estimation results of the accuracy representative, actual results and estimation differences from the actual results are shown in Figure 6.14. The mean error was 0,120234505 (which is close to the error on the training set). The standard deviation was 0,18667105. We consider validation results good.

Task	Agent	Estimation	Result	Error
adult.arff	J48	0,776727779	0,308042	0,468686
adult.arff	RBFNetwork	0,388895804	0,332479	0,056417
adult.arff	OneR	0,243332837	0,434738	0,191405
breast-w.arff	NNge	0,186702384	0,189117	0,002415
breast-w.arff	J48	0,186262062	0,202263	0,016001
breast-w.arff	MultilayerPerceptron	0,130472053	0,160676	0,030204
breast-w.arff	RandomTree	0,180813706	0,18806	0,007247
breast-w.arff	PART	0,169527666	0,193483	0,023955
breast-w.arff	RBFNetwork	0,098833368	0,16391	0,065077
breast-w.arff	OneR	0,207368253	0,270114	0,062745
haberman.arff	NNge	0,218840351	0,527046	0,308206
haberman.arff	J48	0,208131623	0,426859	0,218728
haberman.arff	MultilayerPerceptron	0,254741065	0,413519	0,158778
haberman.arff	RandomTree	0,223016728	0,453078	0,230062
haberman.arff	PART	0,223751771	0,423886	0,200134
haberman.arff	RBFNetwork	0,5202868	0,422565	0,097722
haberman.arff	OneR	0,280444776	0,495074	0,214629
iris.arff	NNge	0,175895561	0,149071	0,026824
iris.arff	J48	0,350507148	0,158559	0,191949
iris.arff	MultilayerPerceptron	0,173929202	0,114573	0,059356
iris.arff	RandomTree	0,182577719	0,185175	0,002597
iris.arff	PART	0,345632005	0,162056	0,183576
iris.arff	OneR	0,389774591	0,2	0,189775
labor.arff	NNge	0,17290295	0,439298	0,266395
labor.arff	J48	0,156961157	0,381398	0,224437
labor.arff	MultilayerPerceptron	0,089412979	0,225341	0,135928
labor.arff	RandomTree	0,187124627	0,336318	0,149193
labor.arff	PART	0,129243608	0,3105	0,181256
labor.arff	RBFNetwork	0,164569296	0,132454	0,032115
labor.arff	OneR	0,254481351	0,439298	0,184816

Task	Agent	Estimation	Result	Error
letter-recognition.arff	J48	0,210489715	0,090255	0,120235
letter-recognition.arff	RandomTree	0,248900172	0,095189	0,153712
letter-recognition.arff	RBFNetwork	0,243904703	0,077127	0,166778
lung-cancer.arff	NNge	0,221843356	0,478714	0,25687
lung-cancer.arff	J48	0,205035648	0,383321	0,178286
lung-cancer.arff	MultilayerPerceptron	0,178754256	0,407859	0,229105
lung-cancer.arff	RandomTree	0,195231692	0,423016	0,227784
lung-cancer.arff	PART	0,164366222	0,393065	0,228699
lung-cancer.arff	RBFNetwork	0,152201513	0,445957	0,293756
lung-cancer.arff	OneR	0,204962277	0,629153	0,424191
magic.arff	J48	0,184620222	0,339662	0,155041
magic.arff	PART	0,195906942	0,332453	0,136546
magic.arff	RBFNetwork	0,431259838	0,351214	0,080046
magic.arff	OneR	0,239329114	0,525281	0,285952
tic-tac-toe.arff	NNge	0,218840351	0,324697	0,105856
tic-tac-toe.arff	J48	0,164686425	0,232614	0,067927
tic-tac-toe.arff	MultilayerPerceptron	0,254741065	0,085667	0,169074
tic-tac-toe.arff	RandomTree	0,223016728	0,356741	0,133724
tic-tac-toe.arff	PART	0,188954094	0,161543	0,027411
tic-tac-toe.arff	RBFNetwork	0,170485221	0,29004	0,119555
tic-tac-toe.arff	OneR	0,235577734	0,548294	0,312716
weather.arff	NNge	0,17530172	0,534523	0,359221
weather.arff	J48	0,184043748	0,267261	0,083217
weather.arff	MultilayerPerceptron	0,136632262	0,342394	0,205761
weather.arff	RandomTree	0,165839595	0,388322	0,222482
weather.arff	PART	0,167521064	0,456435	0,288914
weather.arff	RBFNetwork	0,114129614	0,478396	0,364266
weather.arff	OneR	0,198825049	0,755929	0,557104

Figure 6.14 - Accuracy validation

The error histogram is shown in Figure 6.15.

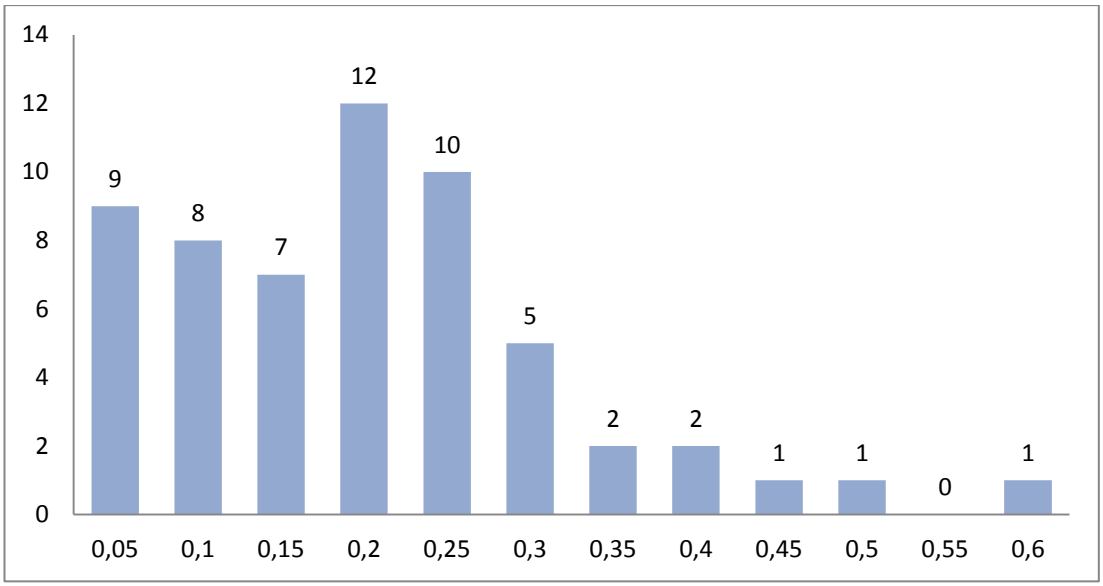


Figure 6.15 - Accuracy validation error histogram

6.3.2 Time validation

The average duration of the validation set results was between 0 and 1994092411. Possible reason for this large range may be that records had not been cleaned. Because of this, we did not use the absolute deviation between estimation and actual results for the error calculation. Instead, we used the order-of-magnitude difference of the estimation and actual results. Duration often vary significantly, even between different problems, and estimation need not to be too precise, therefore we consider this error calculation sufficient. Estimation results of the time representative, actual results and order-of-magnitude differences from the actual results are shown in Figure 6.16. On the majority of validation data the estimation results are again quite good.

Task	Agent	Estimation	Result	Error
adult.arff	J48	486186953,70478	1737857591	1
adult.arff	RandomTree	10,29442	19982	3
adult.arff	RBFNetwork	55,82704	9252	2
adult.arff	OneR	10,29444	3145	2
breast-w.arff	NNge	0,84090	79	1
breast-w.arff	J48	478777695,63746	1641404310	1
breast-w.arff	MultilayerPerceptron	289,75261	5534	1

Task	Agent	Estimation	Result	Error
breast-w.arff	RandomTree	0,00000	1	0
breast-w.arff	PART	260196428,99836	919941747	0
breast-w.arff	RBFNetwork	3,20060	44	1
breast-w.arff	OneR	0,00000	2	0
haberman.arff	NNge	0,00000	4	0
haberman.arff	J48	483726624,84967	1708475065	1
haberman.arff	MultilayerPerceptron	1261,25336	1698	0
haberman.arff	RandomTree	3,87073	1	0
haberman.arff	PART	264782388,33116	916183436	0
haberman.arff	RBFNetwork	1,79248	11	1
haberman.arff	OneR	3,87073	1	0
iris.arff	NNge	0,00000	1	0
iris.arff	J48	484925720,93649	1572264813	1
iris.arff	MultilayerPerceptron	916,74047	867	0
iris.arff	RandomTree	0,00000	1	0
iris.arff	PART	265033387,12327	886760370	0
iris.arff	RBFNetwork	2,72409	13	1
iris.arff	OneR	1,47683	1	0
labor.arff	NNge	27,95228	1	1
labor.arff	J48	487451991,05948	1683475965	1
labor.arff	MultilayerPerceptron	1709,42730	1044	0
labor.arff	RandomTree	0,00000	1	0
labor.arff	PART	252196320,08563	928803830	0
labor.arff	RBFNetwork	17,21355	4	1
labor.arff	OneR	0,00000	2	0
letter-recognition.arff	J48	490492802,04863	1687922526	1
letter-recognition.arff	RandomTree	0,00000	19152	4
letter-recognition.arff	PART	257826814,08912	149245	3
lung-cancer.arff	NNge	3,29421	2	0
lung-cancer.arff	J48	480744351,85136	1683475965	1

Task	Agent	Estimation	Result	Error
lung-cancer.arff	MultilayerPerceptron	166,32509	7156	1
lung-cancer.arff	RandomTree	0,00000	1	0
lung-cancer.arff	PART	327941898,60945	934521305	0
lung-cancer.arff	RBFNetwork	0,00000	6	0
lung-cancer.arff	OneR	0,00000	1	0
magic.arff	J48	481436611,60840	1680764131	1
magic.arff	PART	264002889,56198	1994092411	1
magic.arff	RBFNetwork	9,76430	6747	3
magic.arff	OneR	1,68179	607	2
tic-tac-toe.arff	NNge	0,00000	72	1
tic-tac-toe.arff	J48	480781743,34571	1698719319	1
tic-tac-toe.arff	MultilayerPerceptron	1261,25336	11408	1
tic-tac-toe.arff	RandomTree	0,00000	3	0
tic-tac-toe.arff	PART	256245551,00588	852761416	0
tic-tac-toe.arff	RBFNetwork	5,09241	54	1
tic-tac-toe.arff	OneR	0,00000	1	0
weather.arff	NNge	10,56990	1	1
weather.arff	J48	481436611,60840	1676159155	1
weather.arff	MultilayerPerceptron	887,15185	108	0
weather.arff	RandomTree	3,59491	1	0
weather.arff	PART	263842842,39059	845900441	0
weather.arff	RBFNetwork	7,85357	1	0
weather.arff	OneR	3,59491	1	0

Figure 6.16 - Duration validation

The order-of-magnitude error histogram is shown in Figure 6.17.

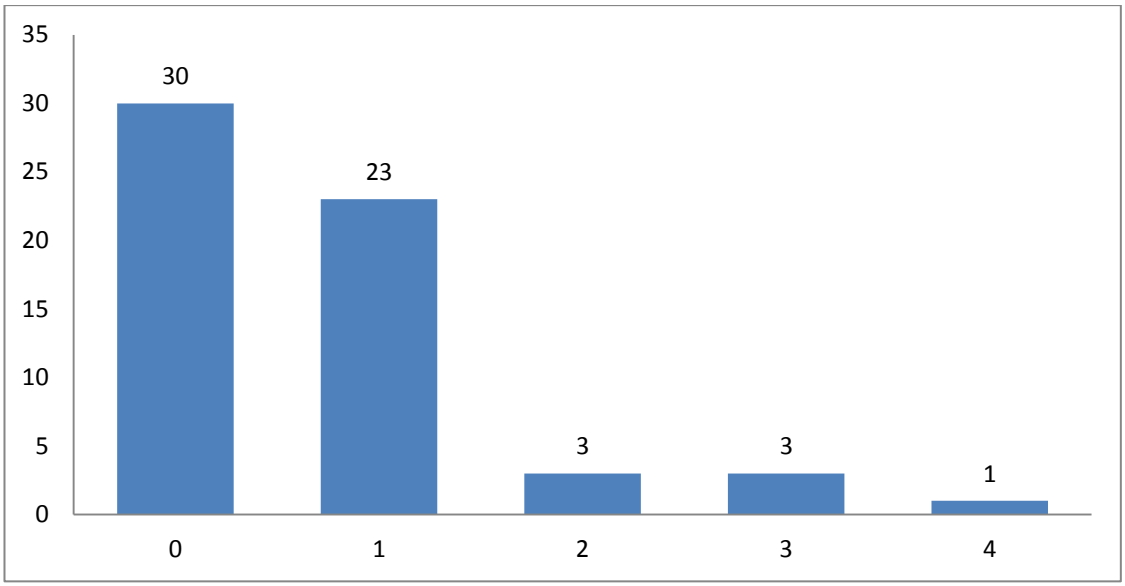


Figure 6.17 - Duration validation error histogram

6.4 Discussion

This section summarizes the results of the experiments and discusses them.

Both accuracy and time estimation experiments were performed. The goal was to obtain trees for accuracy and time estimation. The inputs included previous results of an agent and complexity based on the new task's metadata. The outputs were accuracy and time estimation. For fitness, we used the root mean squared error for the accuracy estimation, and the duration expressed in seconds for the time estimation. Different settings were tested in order to find the best solutions in each type of experiment. Training set containing 43 unique pairs of (agent, task) was reduced to the size of 36 records for accuracy experiments and to the size of 39 records for time experiments after detailed examination. Bloat was encountered during experiments. Proper countermeasures were taken in response. In the accuracy estimation experiments, we did not choose the individual with the best fitness as representative one. Instead, we chose the individual with the best ratio of fitness and size because smaller individuals often generalize better. The error of this individual was 0.03 per test case in average. Test cases' RMSE values were between 0 and 1. We have chosen the individual with the best fitness as a representative in the time estimation. Average estimation error had about ten percent deviation from the test cases.

Representative individuals were validated on 60 pairs from the validation set. The average error on the validation set was 0,120234505 for the accuracy estimation representative. This error is close to the error of the representative on the training set, making for a good result. Order-of-magnitude difference of the duration estimation representative was good for the most agents.

We consider errors of obtained individuals fair. Results suggest that GP have the potential to be successfully used in metalearning. With regard to the sizes of training sets, more training data would be required to obtain more conclusive results.

7 Conclusions

The goal of this thesis was to design and implement an agent capable of estimation of other data mining agents performances in real time. The main goal was divided into smaller subgoals described in section 2.4. This chapter enumerates them.

A metric comparing data mining problems was proposed in section 4.2. Our metric proposed proceeds from a metric already used in the field and improves it by removing the dependency on the order of meta-attributes. The metric was theoretically analysed. An efficient algorithm computing the metric was designed based on the analysis and its time complexity $O(n * \log n)$ was derived. The algorithm was used to compute distances of some widely known problems in chapter 6.

A genetic programming algorithm capable of searching various program spaces was introduced in section 3.2. We have made the decision to split the performance estimation into accuracy and time estimation in section 4.3. The GP goal was to find two trees which would estimate time and accuracy performance of an agent on the new data mining task. We have identified two search spaces, one for accuracy estimation and one for time estimation in section 4.4. Search space utilized the metric for finding tasks similar to the new task, and introduced terminals for obtaining agent's previous performance results on those similar tasks. Metadata were used to propose a terminal estimating complexity of the new task. Various mathematical functions and terminals were also used in the search space. Other functions and terminals were also discussed and the reasons for not introducing them into the domain were stated.

The custom genetic programming framework was introduced in section 5.3. The framework is able to execute wide range of GP experiments, allow parallel fitness computation, and provides tools for viewing and persisting GP entities. Experiments to find accuracy and estimation tree were performed in chapter 6. Bloat was encountered during the experiments (see section 6.1). In response, operators scaling the fitness of the bloated individuals were used. The results of the

Experiments suggest that genetic programming has the potential to be successfully used in metalearning.

An agent architecture encapsulating the estimation trees was proposed in section 4.5. The goal of the architecture is to provide estimation of all agents after receiving a query with a new task. The proposed architecture satisfies all three conditions of the intelligent agent architecture. Middleware for the development and runtime execution of agents - JADE - was used to implement the architecture. Thanks to JADE, it is easy to add the agent into existing multi-agent systems.

8 Future work

This chapter discusses possible future directions of research in the field of genetic optimization of estimation agents.

- More training data: A sufficient amount of training data is crucial for the good performance of all machine learning algorithms. Running the experiments with larger training sets may produce better individuals with better generalization capabilities. A method for generating random samples may be proposed as well.
- More metadata: More metadata may be extracted from the tasks. With more metadata availaible, the metric between data mining problems may yield more accurate results.
- GP algorithm enhancements: GP results strongly depends on the operator settings and on the operators used. It may be beneficial to test different settings of operators and to introduce new genetic operators into the algorithm. More complex GP domains may also improve the best individual obtained. Some of the functions and terminals discussed in 4.4 may also be tested. This direction is the most time and resource consuming.
- Different metric between data mining problems: It may be desirable to compare the results of the agent using different metrics. Time complexities and benefits of various metrics may also be a subject of future research. For example, a metric may be proposed that does not depend on the order of the attributes and may be computed quickly. This benefit is possible because of the linearly ordered categorical and integer attributes (based on the number of attributes and $|Max - Min|$). By removing this assumption we can get more complex metrics. Future work may identify such metrics and compare complexity of algorithms calculating them.
- Time and accuracy preferences: in 4.3 we opted not to use the mechanism of accuracy and time preference exchange. Future work may propose such mechanisms. An agent making assumptions about such preferences may be proposed as well. The best way would be to identify the most common

preferences which would be used unless the inquiring agent requests otherwise.

List of Figures

Bibliography

[1] D. H. Wolpert, "The supervised learning no-free-lunch Theorems," in *In Proc. 6th Online World Conference on Soft Computing in Industrial Applications*, London, Springer-Verlag, 2001, pp. 25-42.

[2] M. Wooldridge, "Intelligent agents," in *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, Cambridge, MIT Press, 1999, pp. 27-77.

[3] S. Russel and P. Norvig, Artificial Intelligence: A Modern Approach, New Jersey: Prentice Hall, 2009.

[4] J. R. Searle, Speech Acts: An Essay in the Philosophy of Language, Cambridge: Cambridge University Press, 1970.

[5] T. Schaul and J. Schmidhuber, "Metalearning," 2010. [Online]. Available: http://www.scholarpedia.org/article/Metalearning. [Accessed 2012].

[6] P. Brazdil, R. Vilalta, C. Giraud-Carrier and C. Soares, "Using Meta-Learning to Support Data Mining," *International Journal on Computational Science & Applications,* vol. 1, pp. 31-45, 2004.

[7] P. Brazdil and C. Soares, "Zoomed Ranking: Selection of Classification Algorithms based on Relevant Performance Information," in *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, London, Springer, 2000, pp. 126-135.

[8] R. Neruda, K. Pešková, O. Kazík and M. Pilát, "Meta Learning in Multi-agent Systems for Data Mining.," in *International Conference on Intelligent Agent Technology*, Los Alamitos, IEEE Computer Society, 2011, pp. 433-434.

[9] C. Darwin, On the Origin of Species by Means of Natural Selection, London: John Murray, 1859.

[10] G. Mendel, "Versuche über Pflanzen-Hybriden," *Verh. Naturforsch. Ver. Brünn,* vol. 4, p. 3–47, 1865.

[11] J. H. Holland, Adaptation in natural and artificial systems, Ann Arbor: University of Michigan Press, 1975.

[12] R. Poli, W. Langdon and N. F. McPhee, A Field Guide to Genetic Programming, Barking: Lulu Enterprises, 2008.

[13] J. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, Cambridge: The MIT Press, 1992.

[14] I. Harvey, D. Cliff and P. Husbands, "Seeing the Light: Artificial Evolution," in *Proceedings of the third international conference on Simulation of adaptive behavior : from animals to animats 3*, Brighton, MIT Press, 1994, pp. 392-401.

[15] W. B. Langdon, "Size Fair and Homologous Tree Crossovers for Tree Genetic Programming," *Genetic Programming and Evolvable Machines,* vol. 1, pp. 95-119, 2000.

[16] R. Poli, "A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming," in *Genetic Programming, Proceedings of EuroGP'2003*, Essex, Springer-Verlag, 2003.

[17] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology,* vol. 48, pp. 443-453, 1970.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations,* vol. 11, no. 1, 2009.

[19] F. Bellifemine, G. Caire, A. Poggi and G. Rimassa, "JADE: A White Paper," Telecom Itali, Parma, 2003.

[20] A. Frank and A. Asuncion, "UCI Machine Learning Repository," University of California, Irvine, School of Information and Computer Sciences, 2010. [Online]. Available: http://archive.ics.uci.edu/m. [Accessed 2012].

APPENDIX A

DVD-ROM

The enclosed DVD contains source codes of the genetic programming framework, the estimation agent and the metadata extraction tool, the MS SQL database with records used for GP experiments training and validation, the extracted metadata, installation files of the .Net framework 4.0, MS SQL Server 2008 and JADE, and serialized state of the best experiments. The structure of the DVD is described in the index.html file in the root directory.